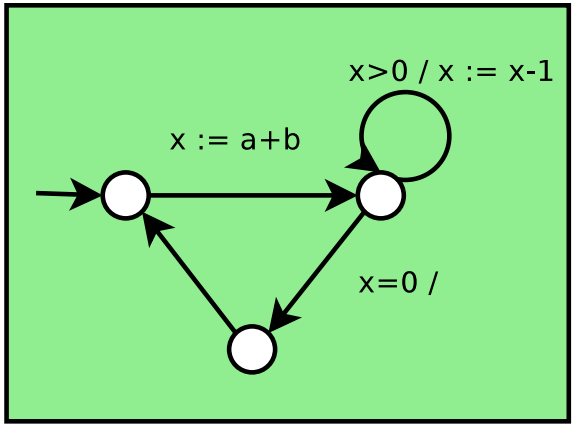
Binary-level Software Analysis

Sébastien Bardin
joint work with Franck Védrine and Philippe Herrmann

CEA-LIST, Laboratoire de Sûreté Logicielle

Binary-level software analysis

Model



Source code

```
int foo(int x, int y) {
 int k= x;
 int c=y;
 while (c>0) do {
   k++;
   c--;}
 return k;
}
```

Assembly

```
_start:
  load  A 100
  add B A
  cmp B 0
  jle label

label:
  move @100 B
```

Executable

```
ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000
```

Benefits of binary code analysis

Advantages over source-level analysis

- executable always available
- no "compiler gap" (security, safety)

New fields of application

- COTS (including libraries)
- mobile code (including malware)
- third-party certification

Benefits of binary code analysis

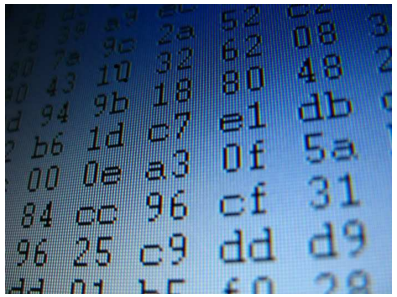
Advantages over source-level analysis

- executable always available
- no "compiler gap" (security, safety)

New fields of application

- COTS (including libraries)
- mobile code (including malware)
- third-party certification

Appealing, but more challenging than source code analysis



Challenges of binary code analysis

D1 : Low-level semantics of data

- machine arithmetic, bit-level operations
- systematic usage of untyped memory [≈ big array]
- ▶ difficult for any state-of-the-art formal technique

D2 : Low-level semantics of control

- no distinction data / instructions, dynamic jumps (`goto A`)
- ▶ no easy syntactic recovery of CFG
- ▶ violate an implicit prerequisite for most formal techniques

D3 : Diversity of architectures and instruction sets

- support for many instructions, modelling issues
- ▶ tedious, time consuming, error prone

Challenges of binary code analysis

D1 : Low-level semantics of data

- machine arithmetic, bit-level operations
- systematic usage of untyped memory [≈ big array]
- ▶ difficult for any state-of-the-art formal technique

D2 : Low-level semantics of control

- no distinction data / instructions, dynamic jumps (`goto A`)
- ▶ no easy syntactic recovery of CFG
- ▶ violate an implicit prerequisite for most formal techniques

D3 : Diversity of architectures and instruction sets

- support for many instructions, modelling issues
- ▶ tedious, time consuming, error prone

Challenges of binary code analysis

D1 : Low-level semantics of data

- machine arithmetic, bit-level operations
- systematic usage of untyped memory [≈ big array]
- ▶ difficult for any state-of-the-art formal technique

D2 : Low-level semantics of control

- no distinction data / instructions, dynamic jumps (`goto A`)
- ▶ no easy syntactic recovery of CFG
- ▶ violate an implicit prerequisite for most formal techniques

D3 : Diversity of architectures and instruction sets

- support for many instructions, modelling issues
- ▶ tedious, time consuming, error prone

Challenges of binary code analysis

D1 : Low-level semantics of data

- machine arithmetic, bit-level operations
- systematic usage of untyped memory [≈ big array]
- ▶ difficult for any state-of-the-art formal technique

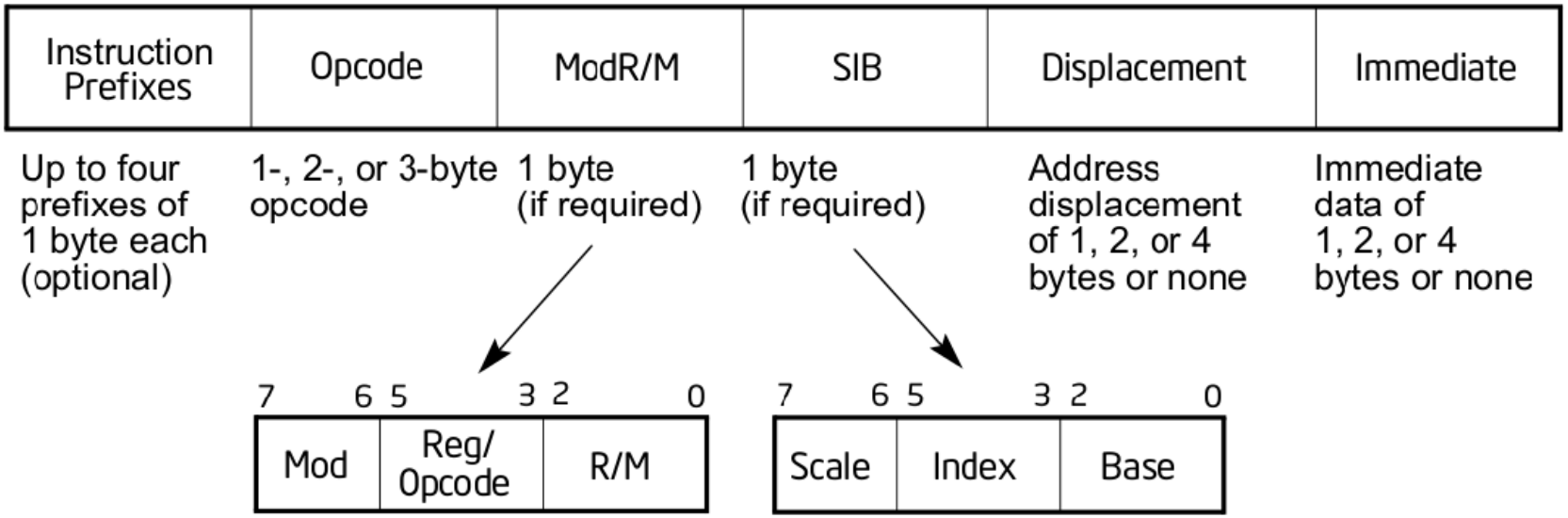
D2 : Low-level semantics of control

- no distinction data / instructions, dynamic jumps (`goto A`)
- ▶ no easy syntactic recovery of CFG
- ▶ violate an implicit prerequisite for most formal techniques

D3 : Diversity of architectures and instruction sets

- support for many instructions, modelling issues
- ▶ tedious, time consuming, error prone

Focus : modelling issues



Focus : modelling issues

32-bits

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
sreg			ES	CS	SS	DS	FS	GS	res.	res.
eee			CR0	invd	CR2	CR3	CR4	invd	invd	invd
eee			DR0	DR1	DR2	DR3	DR4[1]	DR5[1]	DR6	DR7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hex)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[sib]		100	04	0C	14	1C	24	2C	34	3C
disp32		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[sib]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[sib]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
AL/AX/EAX/ST0/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
CL/CX/ECX/ST1/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
DL/DX/EDX/ST2/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
BL/BX/EBX/ST3/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
AH/SP/ESP/ST4/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
CH/BP/EBP/ST5/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
DH/SI/ESI/ST6/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
BH/DI/EDI/ST7/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

16-bits

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
sreg			ES	CS	SS	DS	FS	GS	res.	res.
eee			CR0	invd	CR2	CR3	CR4	invd	invd	invd
eee			DR0	DR1	DR2	DR3	DR4[1]	DR5[1]	DR6	DR7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hex)							
[BX+SI]	00	000	00	08	10	18	20	28	30	38
[BX+DI]		001	01	09	11	19	21	29	31	39
[BP+SI]		010	02	0A	12	1A	22	2A	32	3A
[BP+DI]		011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
disp16		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX+SI]+disp8	01	000	40	48	50	58	60	68	70	78
[BX+DI]+disp8		001	41	49	51	59	61	69	71	79
[BP+SI]+disp8		010	42	4A	52	5A	62	6A	72	7A
[BP+DI]+disp8		011	43	4B	53	5B	63	6B	73	7B
[SI]+disp8		100	44	4C	54	5C	64	6C	74	7C
[DI]+disp8		101	45	4D	55	5D	65	6D	75	7D
[BP]+disp8		110	46	4E	56	5E	66	6E	76	7E
[BX]+disp8		111	47	4F	57	5F	67	6F	77	7F
[BX+SI]+disp16	10	000	80	88	90	98	A0	A8	B0	B8
[BX+DI]+disp16		001	81	89	91	99	A1	A9	B1	B9
[BP+SI]+disp16		010	82	8A	92	9A	A2	AA	B2	BA
[BP+DI]+disp16		011	83	8B	93	9B	A3	AB	B3	BB
[SI]+disp16		100	84	8C	94	9C	A4	AC	B4	BC
[DI]+disp16		101	85	8D	95	9D	A5	AD	B5	BD
[BP]+disp16		110	86	8E	96	9E	A6	AE	B6	BE
[BX]+disp16		111	87	8F	97	9F	A7	AF	B7	BF
AL/AX/EAX/ST0/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
CL/CX/ECX/ST1/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
DL/DX/EDX/ST2/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
BL/BX/EBX/ST3/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
AH/SP/ESP/ST4/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
CH/BP/EBP/ST5/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
DH/SI/ESI/ST6/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
BH/DI/EDI/ST7/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

Focus : modelling issues

32-bits

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
sreg			ES	CS	SS	DS	FS	GS	res.	res.
eee			CR0	invd	CR2	CR3	CR4	invd	invd	invd
eee			DR0	DR1	DR2	DR3	DR4[1]	DR5[1]	DR6	DR7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hex)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[sib]		100	04	0C	14	1C	24	2C	34	3C
disp32		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[sib]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
DH/SI/ESI/ST6/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
BH/DI/EDI/ST7/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

16-bits

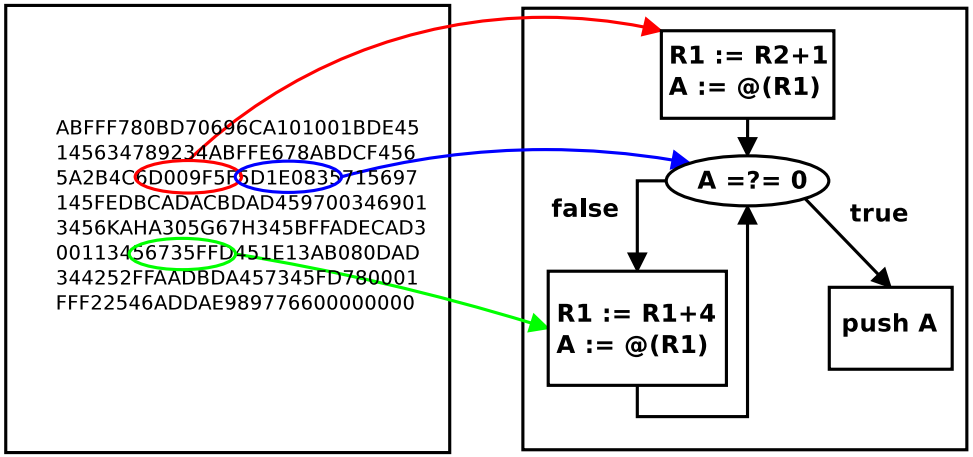
r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
sreg			ES	CS	SS	DS	FS	GS	res.	res.
eee			CR0	invd	CR2	CR3	CR4	invd	invd	invd
eee			DR0	DR1	DR2	DR3	DR4[1]	DR5[1]	DR6	DR7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hex)							
[BX+SI]	00	000	00	08	10	18	20	28	30	38
[BX+DI]		001	01	09	11	19	21	29	31	39
[BP+SI]		010	02	0A	12	1A	22	2A	32	3A
[BP+DI]		011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
disp16		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX+SI]+disp8	01	000	40	48	50	58	60	68	70	78
[BX+DI]+disp8		001	41	49	51	59	61	69	71	79
[BP+SI]+disp8		010	42	4A	52	5A	62	6A	72	7A
[BP+DI]+disp8		011	43	4B	53	5B	63	6B	73	7B
[SI]+disp8		100	44	4C	54	5C	64	6C	74	7C
[DI]+disp8		101	45	4D	55	5D	65	6D	75	7D
[BP]+disp8		110	46	4E	56	5E	66	6E	76	7E
					57	5F	67	6F	77	7F
					90	98	A0	A8	B0	B8
					91	99	A1	A9	B1	B9
					92	9A	A2	AA	B2	BA
					93	9B	A3	AB	B3	BB
					94	9C	A4	AC	B4	BC
					95	9D	A5	AD	B5	BD
					96	9E	A6	AE	B6	BE
					97	9F	A7	AF	B7	BF
					D0	D8	E0	E8	F0	F8
					D1	D9	E1	E9	F1	F9
					D2	DA	E2	EA	F2	FA
					D3	DB	E3	EB	F3	FB
					D4	DC	E4	EC	F4	FC
					D5	DD	E5	ED	F5	FD
DH/SI/ESI/ST6/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
BH/DI/EDI/ST7/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

- more than 1,000 instructions
  still $\approx$ 400 if no float, no interrupts, etc.
- many side-effects, prefixes, ...

Focus : safe CFG recovery problem

Input

- an executable code (array of bytes)
- an initial address
- a basic decoder : file $\times$ address $\mapsto$ instruction $\times$ size



Output : (surapproximation) of the program CFG

Focus : safe CFG recovery problem

The successor instructions can often be identified syntactically

- ⟨ `addr: move a b` ⟩ → successor at `addr+size+1`
- ⟨ `addr: goto 100` ⟩ → successor at `100`
- ⟨ `addr: ble 100` ⟩ → successors at `100` and `addr+size+1`

Focus : safe CFG recovery problem

The successor instructions can often be identified syntactically

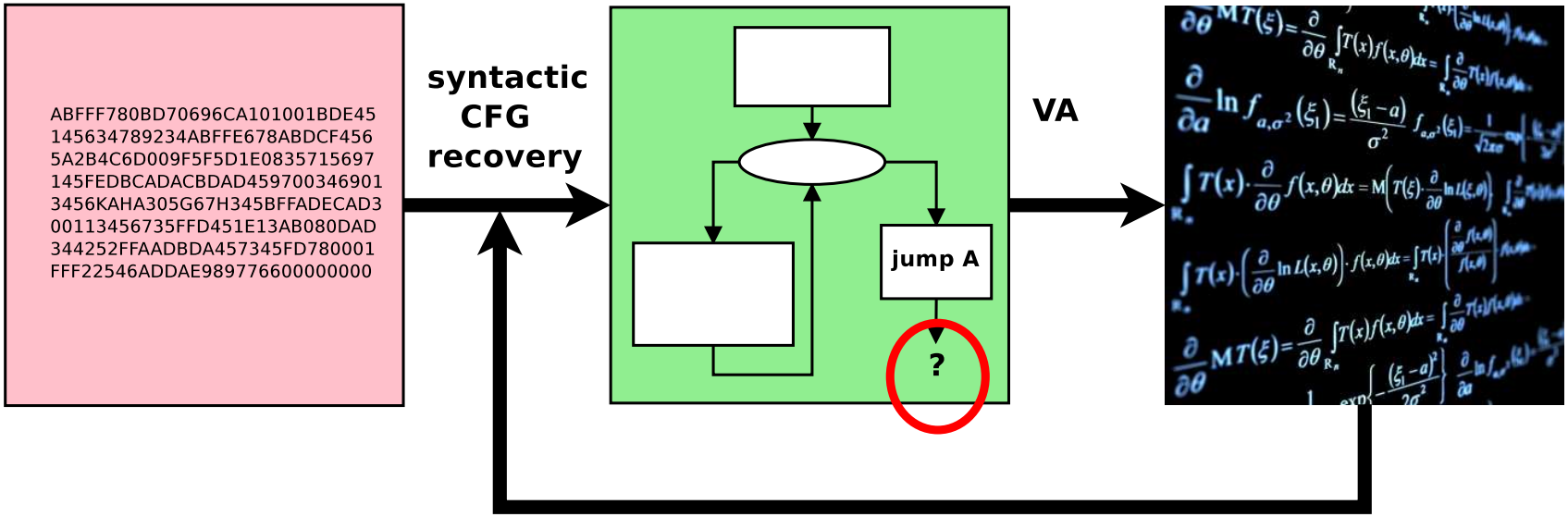
- ⟨ `addr: move a b` ⟩ → successor at `addr+size+1`
- ⟨ `addr: goto 100` ⟩ → successor at `100`
- ⟨ `addr: ble 100` ⟩ → successors at `100` and `addr+size+1`

But not always : successor of ⟨`addr: goto a`⟩ ?

Challenge = compute a superset of all targets of each jump

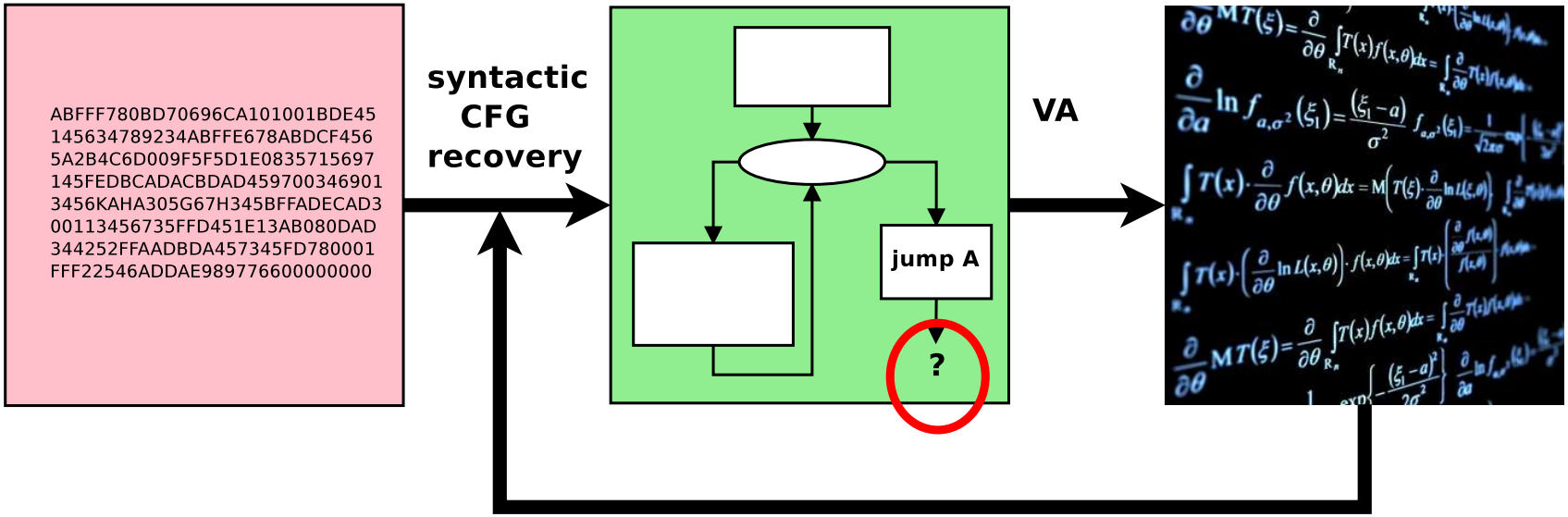
Focus : safe CFG recovery problem

Need to combine syntactic CFG recovery with value analysis (VA)



Focus : safe CFG recovery problem

Need to combine syntactic CFG recovery with value analysis (VA)



"Chicken and egg" problem

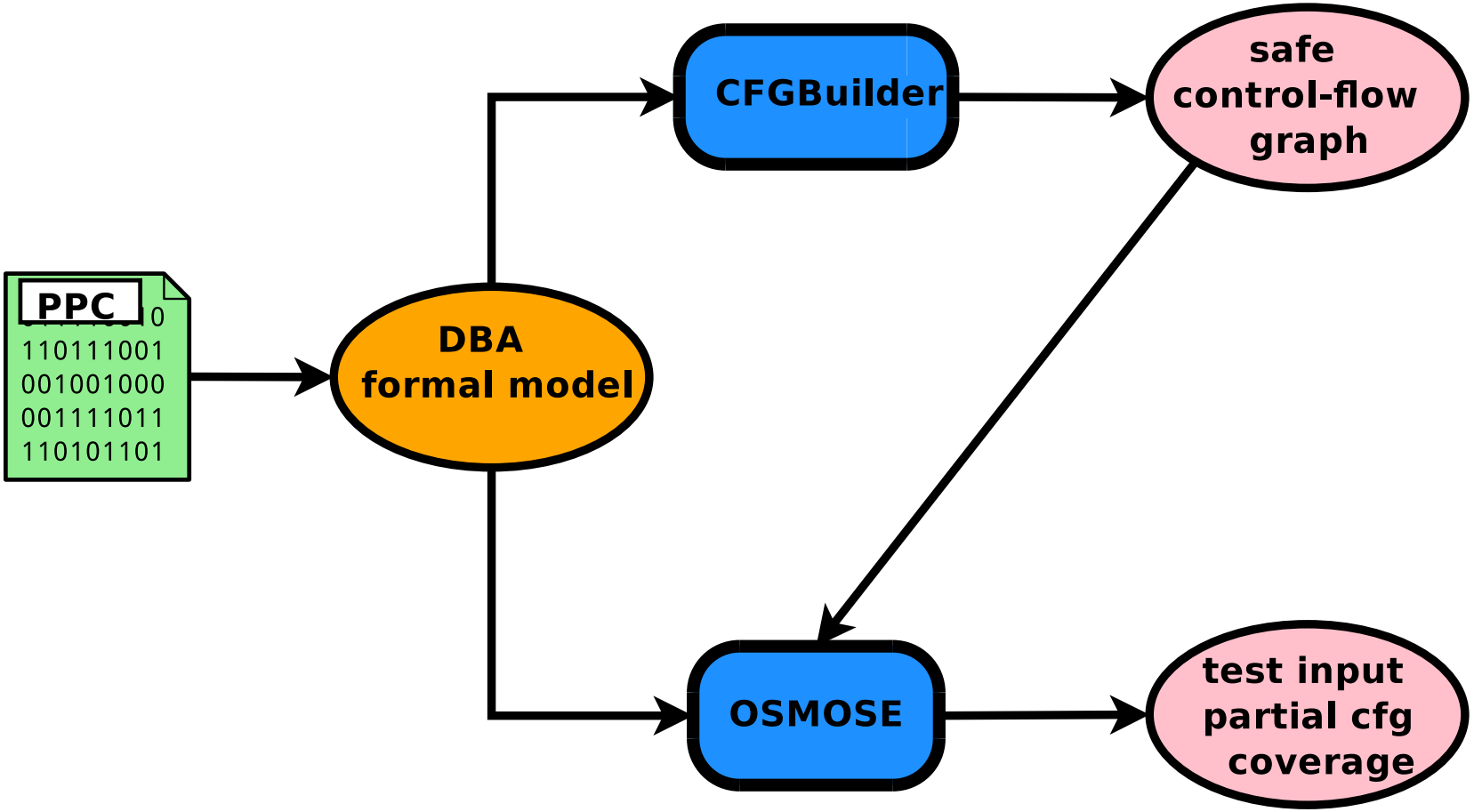
VA imprecise on `goto A`

$\mapsto$ too many instructions / branches added to CFG

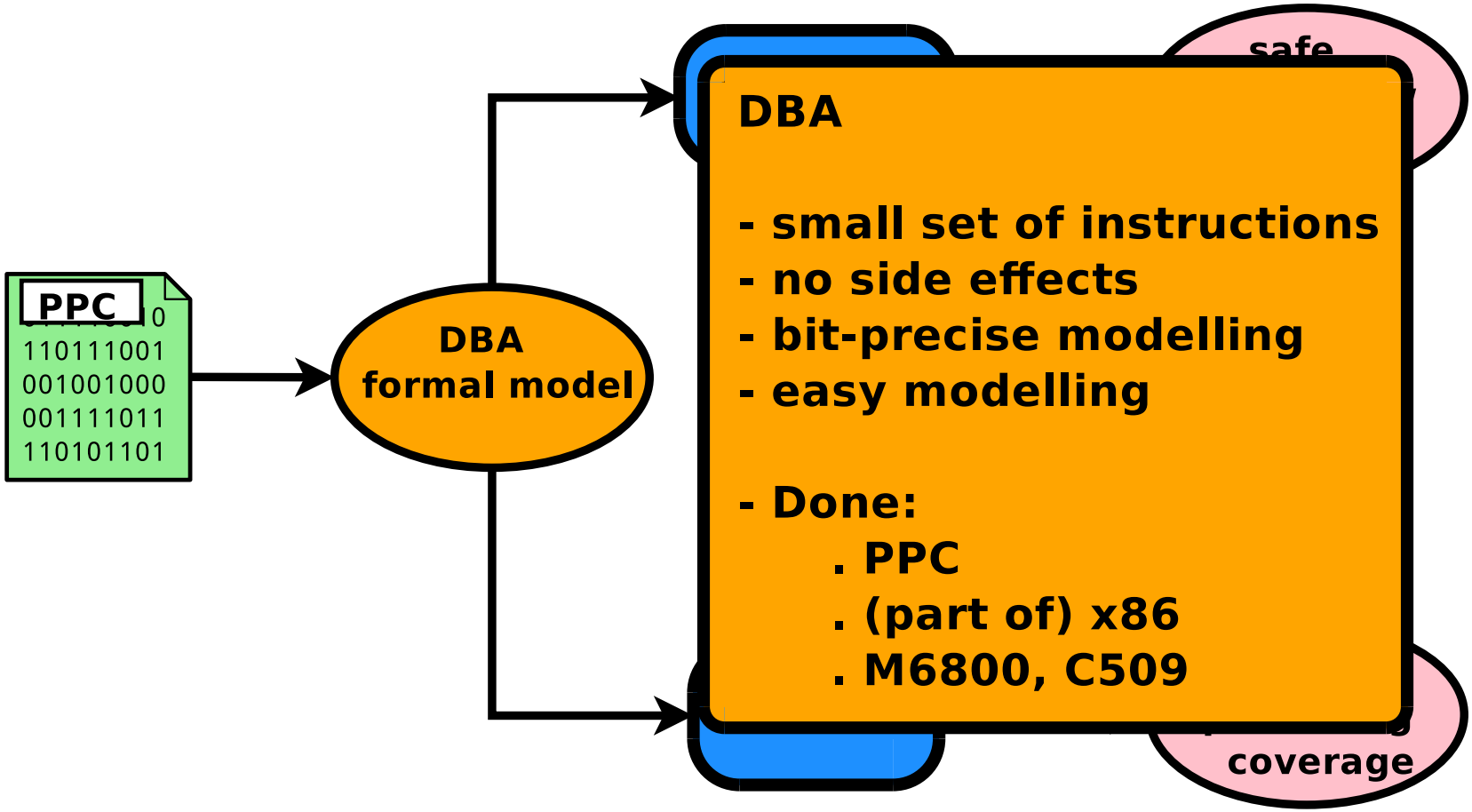
$\mapsto$ more propagation / imprecision in VA

$\mapsto$ VA **more** imprecise on `goto A` $\mapsto$ and so on

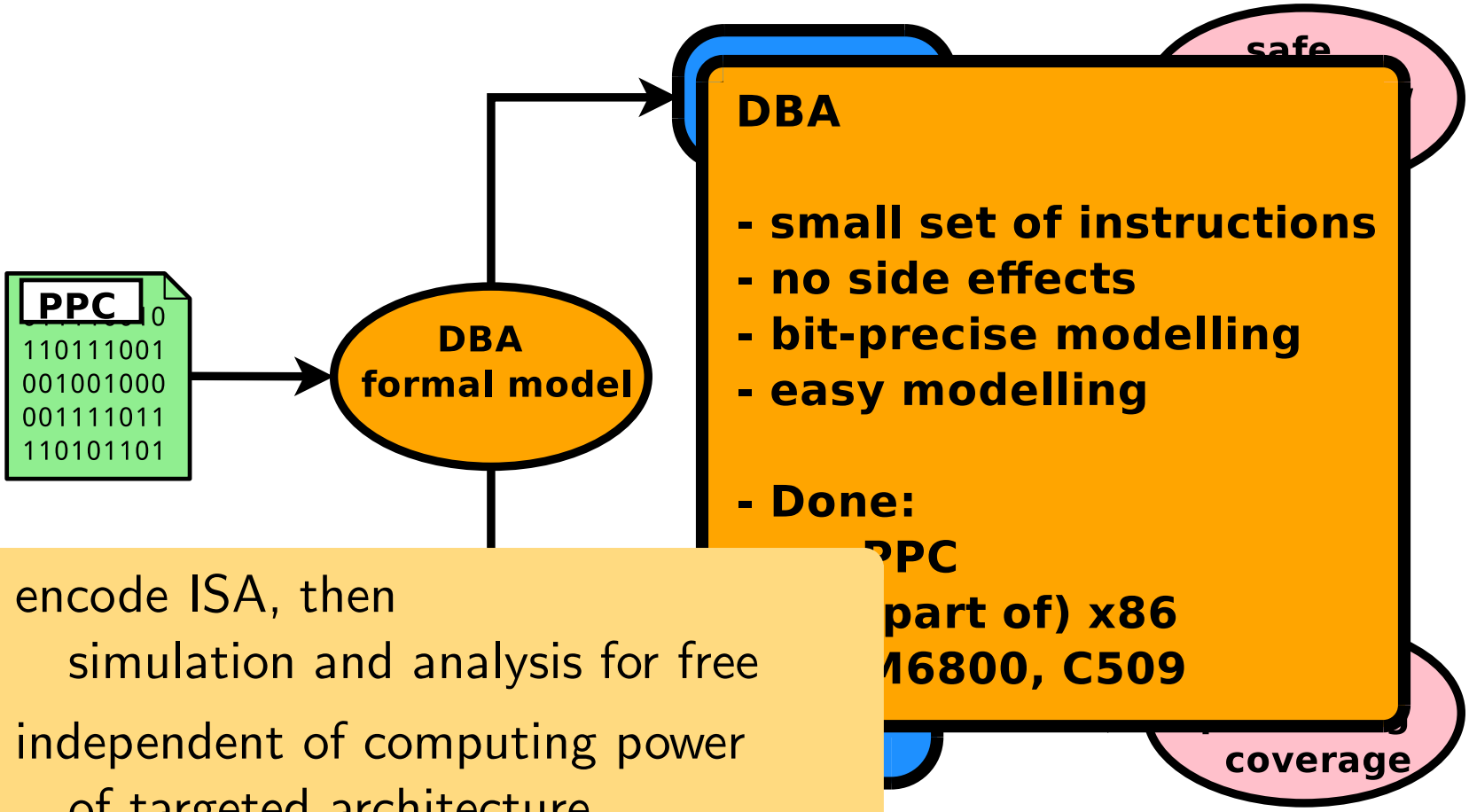
Binary-level program analysis at CEA LSL



Binary-level program analysis at CEA LSL

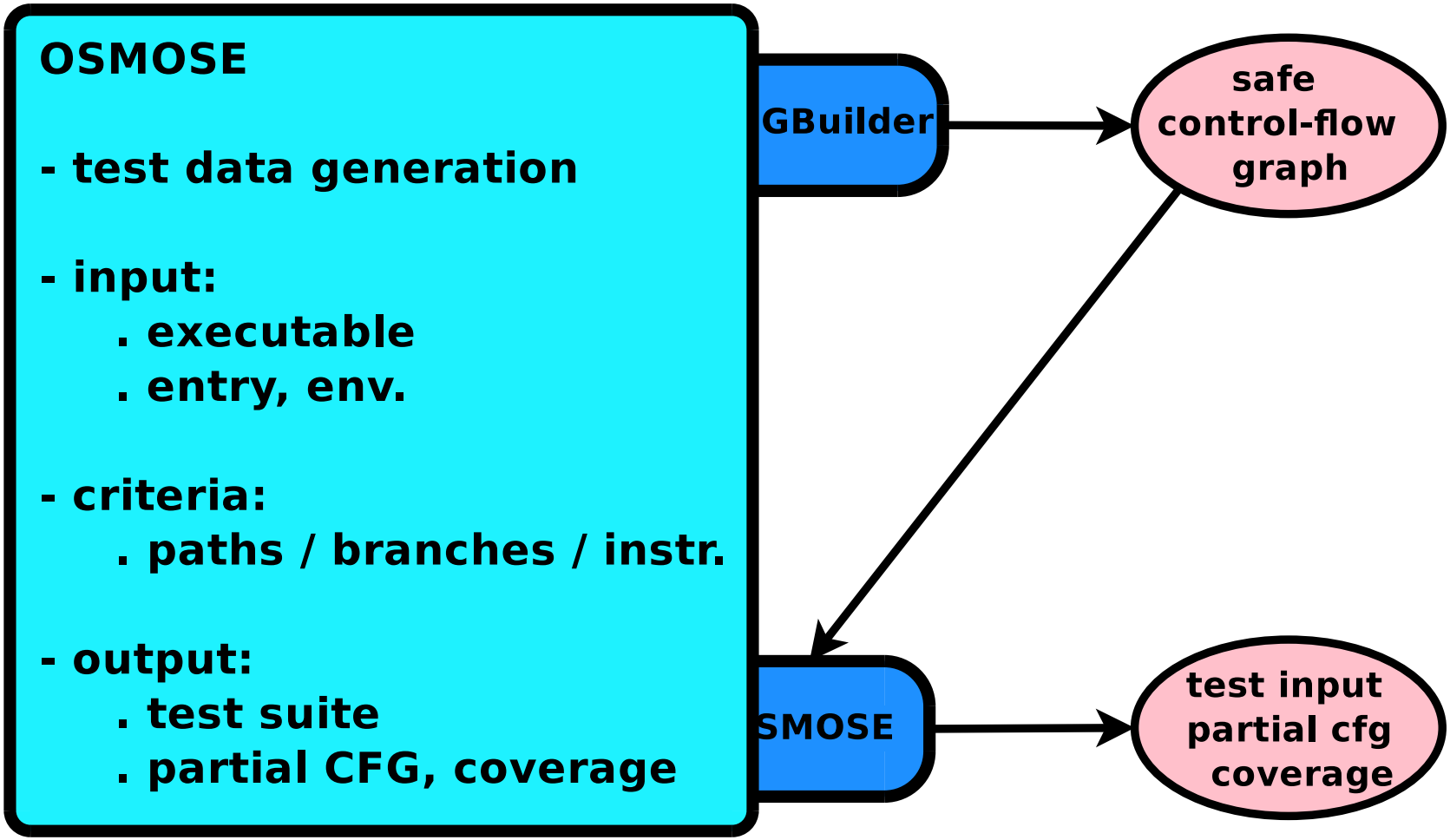


Binary-level program analysis at CEA LSL

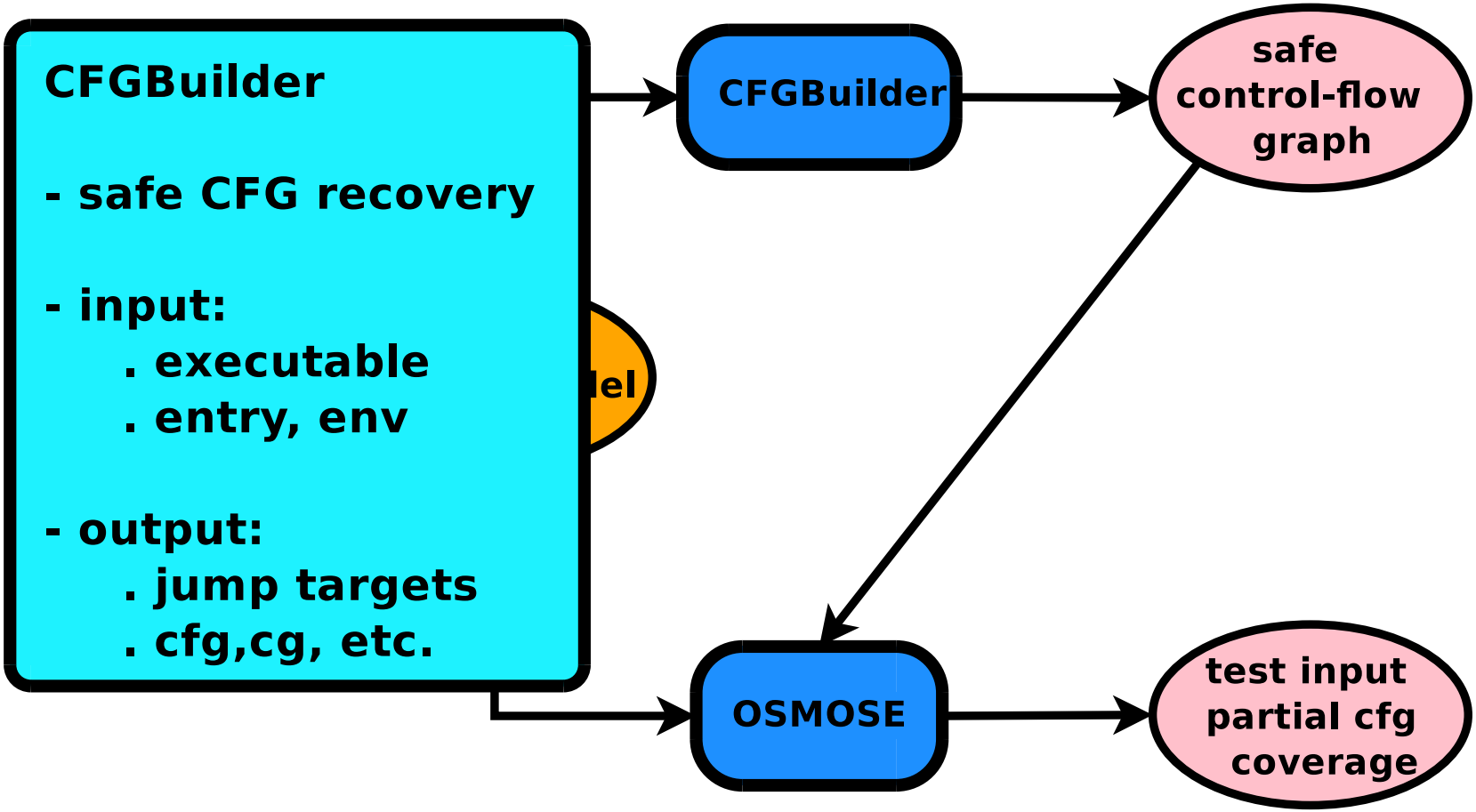


- encode ISA, then
  simulation and analysis for free
- independent of computing power
  of targeted architecture

Binary-level program analysis at CEA LSL



Binary-level program analysis at CEA LSL



Key Technologies

OSMOSE : Dynamic Symbolic Execution [ICST-08, STVR-11]

- exploration of all (bounded) paths of the program
- bit-precise constraint solving [TACAS-10]
- symbolic reasoning to discover new dynamic targets [STVR-11]
- path pruning optimisations [ICST-09, ICST-14]

DBA formal model [CAV-11]

CFGBuilder : Refinement-based analysis [VMCAI-11]

- static analysis through abstract interpretation
- abstract domain = k-sets (finite sets of at most k constants)
- the size is controlled by an iterative refinement mechanism

Outline

Dynamic Bitvector Automata

Main design ideas [CAV 11, with LABRI]

- small set of instructions
- no side-effect
- concise and natural modelling of common ISAs
- low-level enough to allow bit-precise modelling
- standalone model : do not need any info on archi
- try to be "analysis"-agnostic

Can model : instruction overlapping, return address smashing, endianness, overlapping memory read/write

Limitations : (strong) no self-modifying code, (weak) ~~no dynamic memory allocation~~, no FPA

Dynamic Bitvector Automata (2)

Basis

- bitvector variables and arrays of bytes
- all bv sizes statically known
- standard operations from bitvector arithmetic
- some instructions are labelled by addresses

Dynamic Bitvector Automata (2)

Basis

- bitvector variables and arrays of bytes
- all bv sizes statically known
- standard operations from bitvector arithmetic
- some instructions are labelled by addresses

Instructions

- `lhs := rhs, goto addr`
- `goto addr`
- `ite(cond)? goto addr : goto addr'`
- `goto expr`

Dynamic Bitvector Automata (2)

Basis

- bitvector variables and arrays of bytes
- all bv sizes statically known
- standard operations from bitvector arithmetic
- some instructions are labelled by addresses

Conditions

- any expr evaluating to a bv of size 1
- including : `expr` $\{<_{u,s}, \leq_{u,s}, =, \neq, \geq_{u,s}, >_{u,s}\}$ `expr`

Dynamic Bitvector Automata (2)

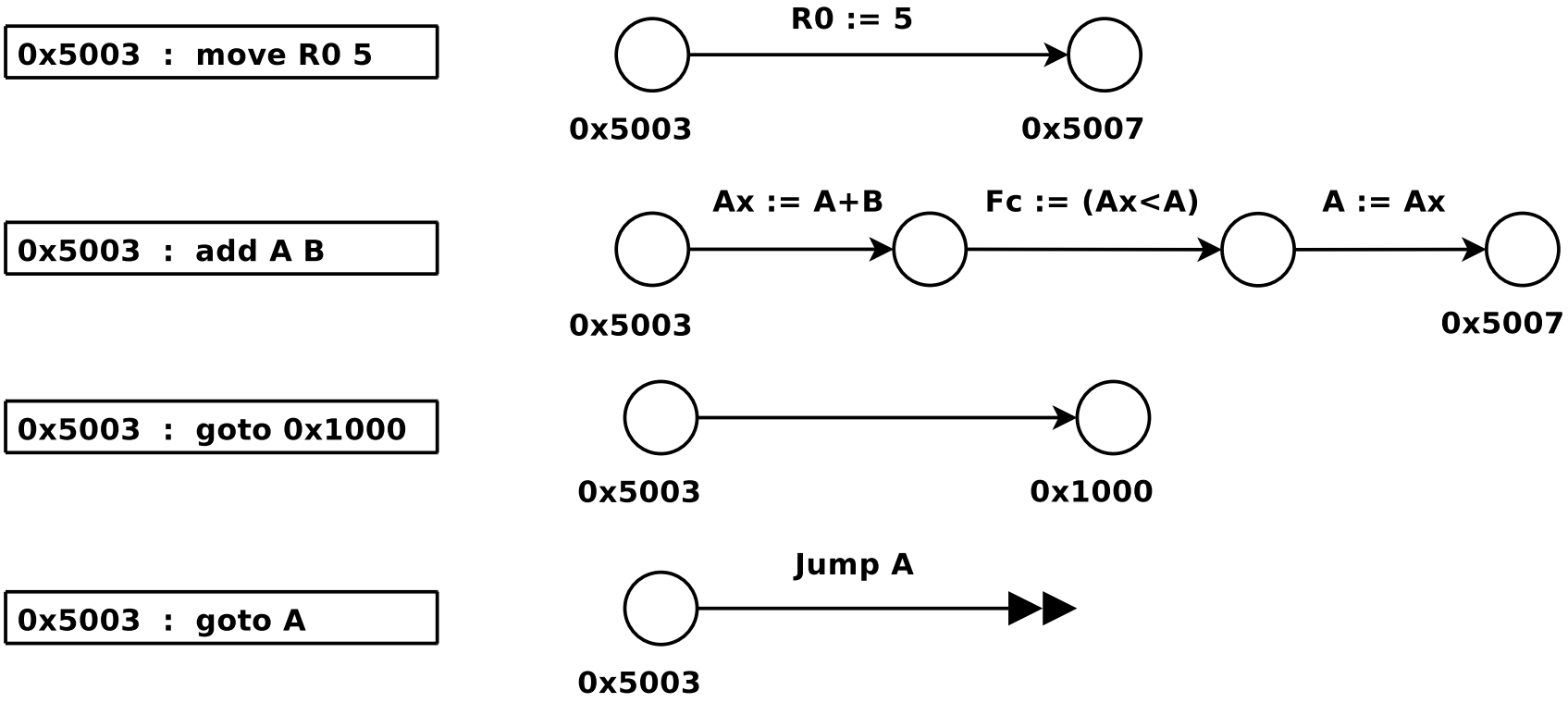
Basis

- bitvector variables and arrays of bytes
- all bv sizes statically known
- standard operations from bitvector arithmetic
- some instructions are labelled by addresses

Expressions

- `0xFF10<16>`, `X<size>`
- `@(expr,` $\overrightarrow{k}$`)`, `@(expr,` $\overleftarrow{k}$`)`
- `expr{i .. j}`, $\texttt{ext}_{u,s}$`(expr,n)`
- `expr` $\{<_{u,s}, \leq_{u,s}, =, \neq, \geq_{u,s}, >_{u,s}\}$ `expr`
- `expr` $\{+, -, \times, /_{u,s}, \%_{u,s}\}$ `expr`
- `expr` $\{\wedge, \vee, \oplus\}$ `expr`, `!expr`
- `expr` $\{<<, >>_{u,s}, ::\}$ `expr`

Modelling with DBA



no procedure calls, only jumps

- `return` becomes `jump @SP`

Fresh news

More (high-level) constructs

- malloc, free
- nondet(size)
- assert(cond), assume(cond)
- stop $<$ ok, ko, none $>$

Region-based memory model [à la CompCert]

- regions are separated parts of memory
- `Constant, Stack, Malloc(id)`
- ease of analysis, but many operations between regions are forbidden
- note : flat model can be encoded

Permissions R,W,X associated to regions

Outline

OSMOSE : test data generation

Goal = automatic test data generation [assume an external oracle]

- under-approximation analysis
- computes witnesses of reachability
- cannot prove invariance

Input : executable, entry point, initial state [including volatile]

Output

- a set of pairs $<$ input, intended execution path $>$
- (under-approximated) CFG
- (under-approximated) coverage measure

[First dse tool over exec. code, with SAGE - Godefroid-08]

Core concept : Dynamic Symbolic Execution

Symbolic Execution (assume a program P)

- choose a path π of P
- compute a *path predicate* φ_π :
  $$\mathtt{v} \models \varphi_\pi \Rightarrow \mathtt{P(v)} \text{ follows } \pi$$ [wpre, spost]
- solve φ_π for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, π $>$, update coverage
- loop until nothing more to cover

Old idea [King-70], but requires powerful solvers

Renew interest in the mid 2000's [dart, cute, pathcrawler, sage, exe, etc.]

Core concept : Dynamic Symbolic Execution (2)

Dynamic Symbolic Execution [Williams+ 04, Godefroid+ 05]

- interleave concrete and symbolic executions
- drive the search towards feasible paths for free
- give hints for relevant under-approximations [concretization]

Core concept : Dynamic Symbolic Execution (2)

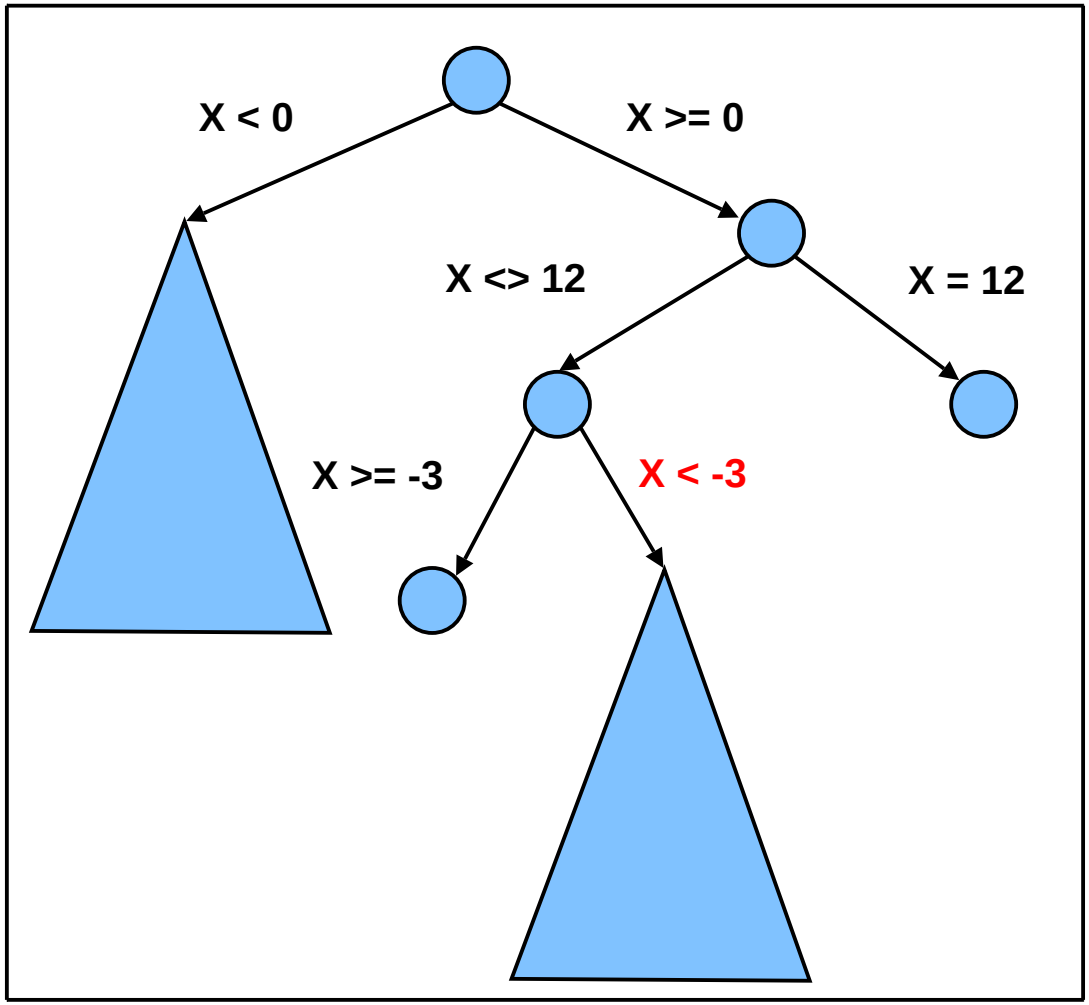
Dynamic Symbolic Execution [Williams+ 04, Godefroid+ 05]

- interleave concrete and symbolic executions
- drive the search towards feasible paths for free
- give hints for relevant under-approximations [concretization]

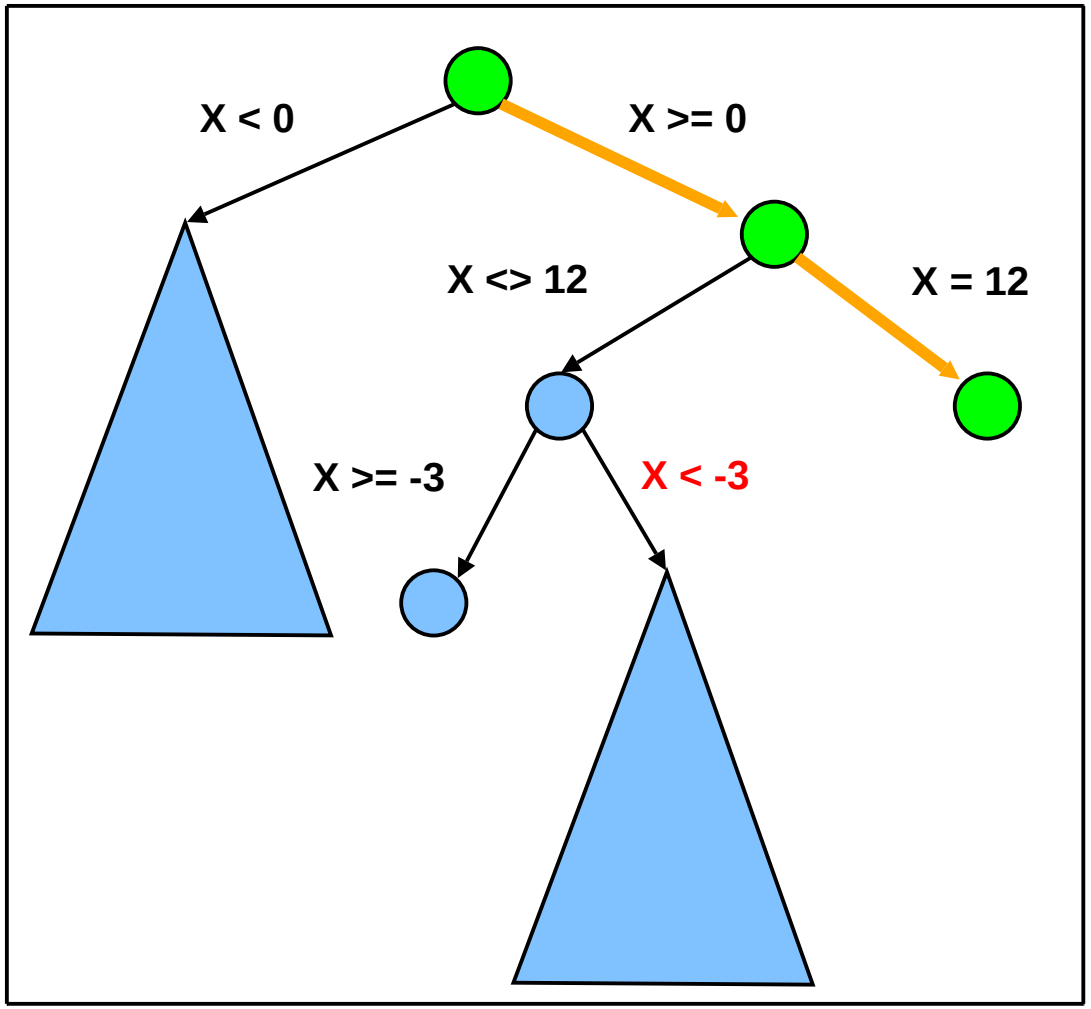
Example of concretization

- suppose instruction `X := A * B`, but only LIA available
- need a proper underapproximation
- suppose another exec. pass through same instr., with `A = 5`
- get the following underapprox : $X = 5 \times B$

Follows feasible paths for free

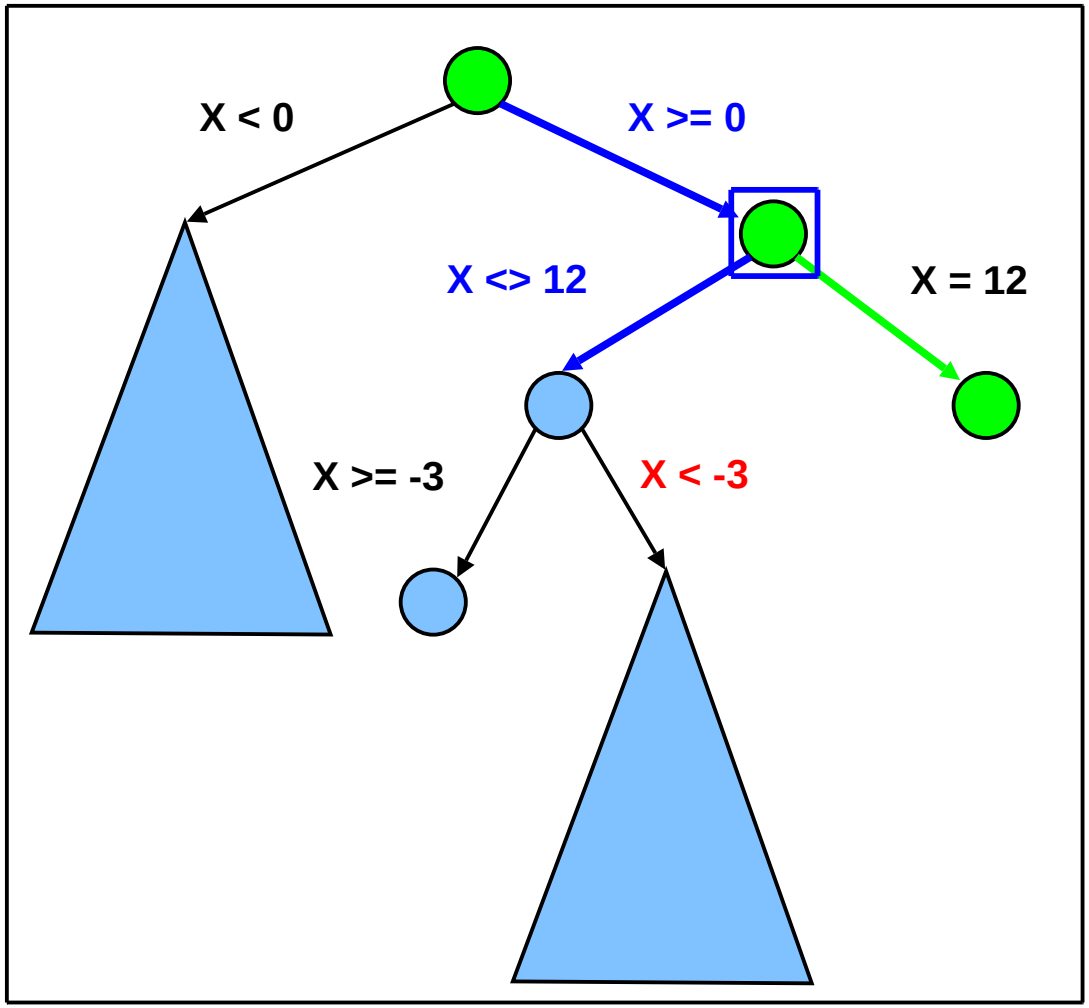


Follows feasible paths for free



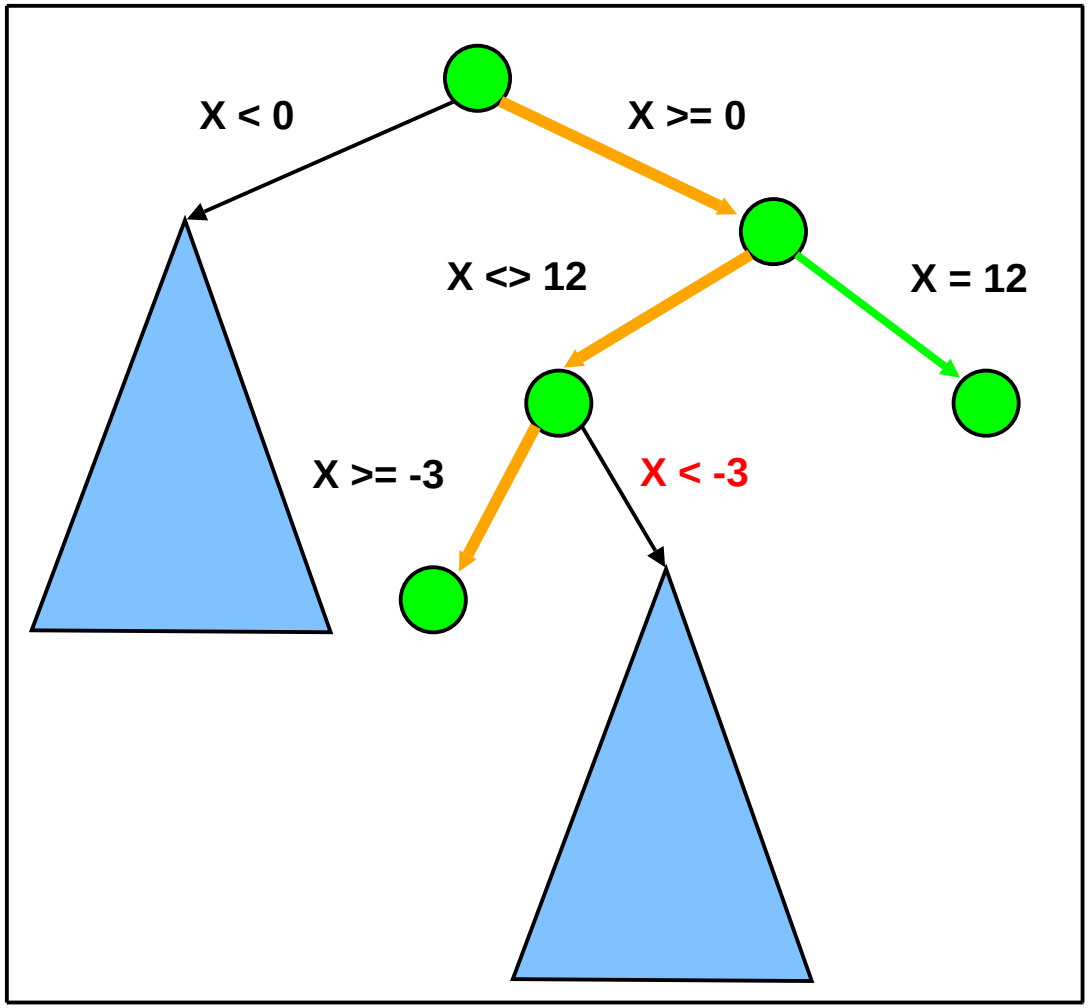
dynamic run with (arbitrary) X=12

Follows feasible paths for free



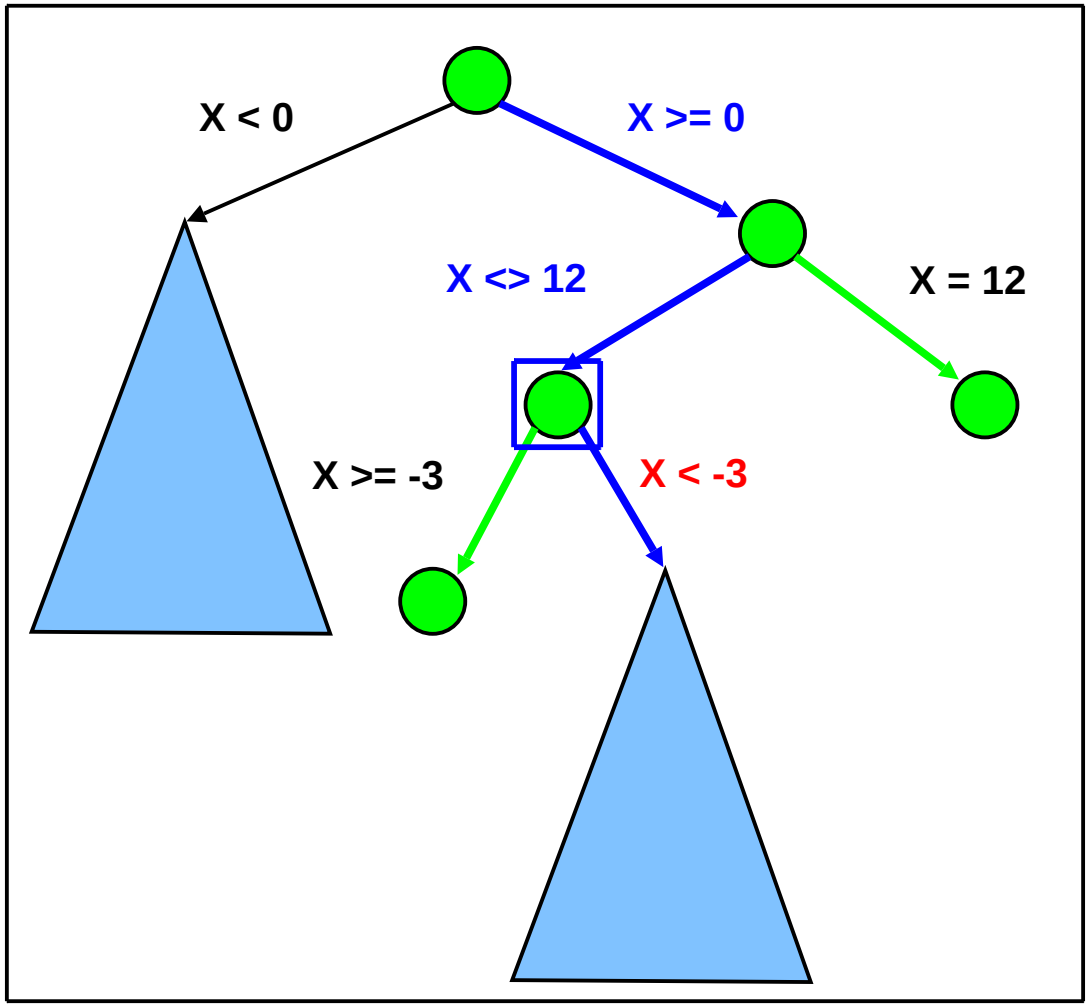
dynamic run with (arbitrary) X=12
backtrack + solve, X = 5

Follows feasible paths for free



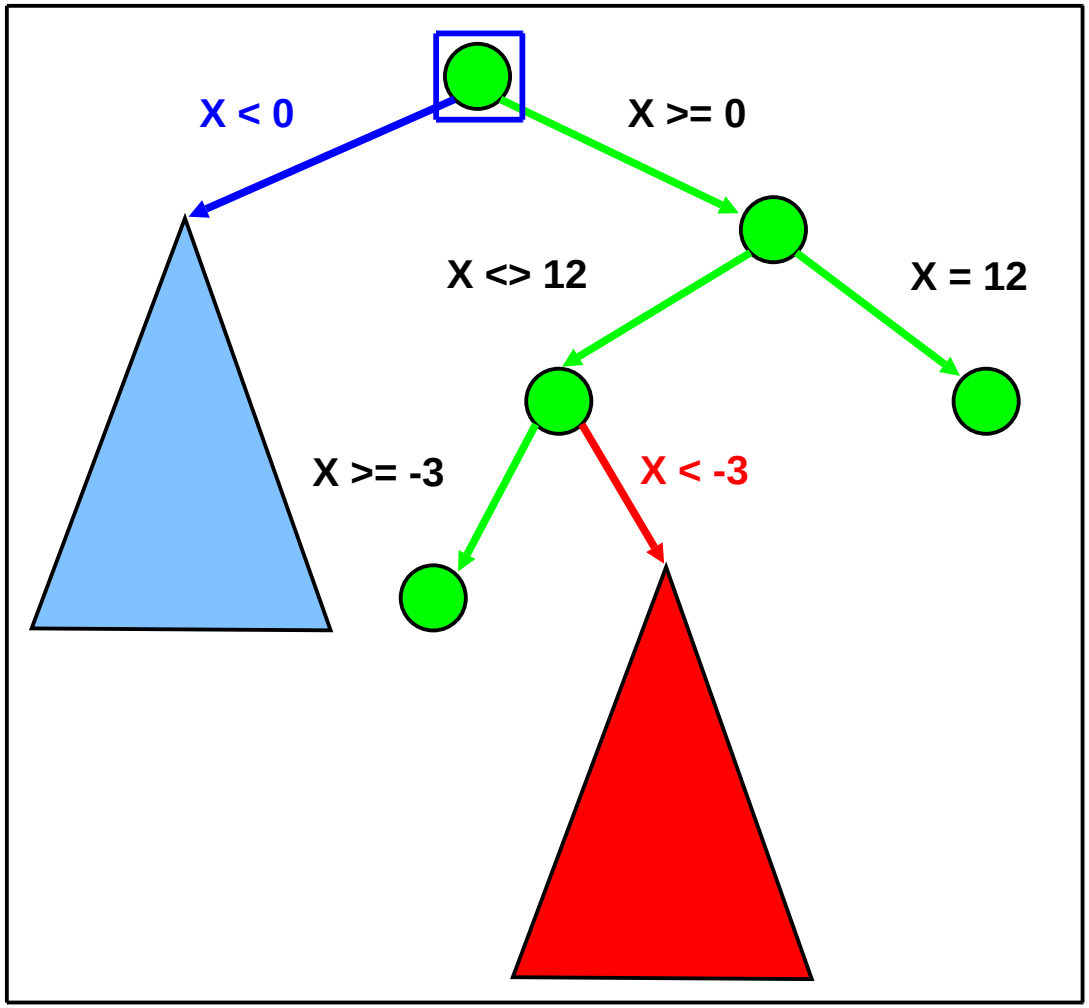
dynamic run with (arbitrary) X=12
backtrack + solve, X = 5
dynamic run with X=5

Follows feasible paths for free



dynamic run with (arbitrary) X=12
backtrack + solve, X = 5
dynamic run with X=5
backtrack + solve, unsat

Follows feasible paths for free



GOOD : never search inside the unfeasible path space

DSE for binary code

Low-level data

- bit-level operation : use bitvector arithmetic ✓
- flat memory model : well, array theory ... ● [concretization may help]

Low-level control

- issue for completeness, not correctness
- can adapt DSE for smart (partial) CFG recovery

DSE for (partial) CFG recovery

Concrete execution allows partial CFG recovery

Symbolic reasoning can be used too [STVR 11]

- assume a prefix π finishing on `jump e`
- assume we already know targets $t_1, \ldots, t_k$
- solve $\varphi_\pi \wedge \bigwedge_{i=1}^{k} e \neq t_i$
- a solution will lead to a new jump target ! !

All targets recovered by OSMOSE are truly reachable !

Optimisations

More efficient solving (even with blackbox solver)

- preprocessing [constant propag, equality propag, variable removal]
- partial reuse of solution of φ_π for solving $\varphi_{\pi\cdot\sigma}$
  transform $\varphi_{\pi\cdot\sigma} = \varphi_\pi \wedge \varphi_\sigma$
  into $\varphi_{\pi\cdot\sigma} = (\varphi_1 \wedge \varphi_2) \wedge \varphi_\sigma$
  s.t. φ_1 does not share any var with φ_2 and φ_σ
- split φ into independent subformulas $\varphi_1 \wedge \varphi_2$

Lower path explosion

- prune paths which cannot reach uncovered items [ICST 09]
- smarter searches than DFS (faster coverage) [QSIC 13]
- efficient handling of coverage objectives [ICST 14]

New features

- generic search engine
- search directives (user-guidance)
- test suite replay and completion
- output of concrete and symbolic states
- specification of dynamic targets
- (a bit of) goal-oriented testing

Experiments

- automatic unit testing of medium-sized aircraft application
- full testing of a small (but tricky) aircraft application
- testing and comprehension of a third-party program
- experimental comparison of source vs binary coverage criteria

First case-study

Medium-size aircraft program (Sagem)

- 30,000 instructions, 250 functions
- max calldepth $= 10$

Goal : unit testing, no expert guidance

Results

- good coverage results for procedures with low height in the call graph (even with 2,000 instructions)
- tested on 40 functions with call-depth ≤ 4 :
  - full cover for 31 functions (in less than a few minutes)
  - bad cover ($< 50\%$) for only 5 functions
- robustness issue with higher-level procedures

Results (2)

name	I	Br	Osmose cover	Osmose time	Osmose #tests	random cover	random time
aircraft0	237	36	100%	10	19	40%	20
aircraft1	290	140	98%	60	43	64%	100
aircraft2	201	72	100%	10	37	35%	20
aircraft3	977	190	50%	60	3	96%	60
aircraft4	2347	500	87%	600	15	68%	600
aircraft5	121 4103	2 509	100%	1	2	100%	10
aircraft6	250 425	18 34	94%	100	9	83%	120
aircraft7	506 15640	20 2790	80%	20	4	75%	500
aircraft8	957 30969	14 4952	14%	10	3	50%	500
aircraft9	627 31793	74 5034	77%	600	12	63%	600

Time in sec. Random tests : 1000 tests - unit testing

Second case-study

Small program (17 procedures and 2,600 instructions), SAGEM

Goal = full testing from the program entry point

Program recognized hard to cover by testing teams

- random testing or DFS-DSE stuck to 50% coverage
- many infeasible paths
- huge search space :
  - one loop must be unrolled $\geq$ 380 times
  - artifical paths due to read-loops on volatile memory

Second case-study

Small program (17 procedures and 2,600 instructions), SAGEM

Goal = full testing from the program entry point

Program recognized hard to cover by testing teams

Approach

- search directives (main loop, read-loops)
- combination of MinCall-BFS and MinCall-DFS

Results

- 100% coverage of 15/17 procedures
- 50% coverage of 2 "library" procedures
- several uncovered branches have been shown to be uncoverable (in progress)

Other experiments

Control-command program written in assembly language (EdF)

- Third-party software, sparse documentation
- Small program, but required very long sequences
- Complete functional tests

+ help understand the code (unfeasible branches, entries, etc.)
+ help to pinpoint problems in doc (ack. by vendor)

Comparison of binary coverage vs source coverage [S. Labbé, EdF]

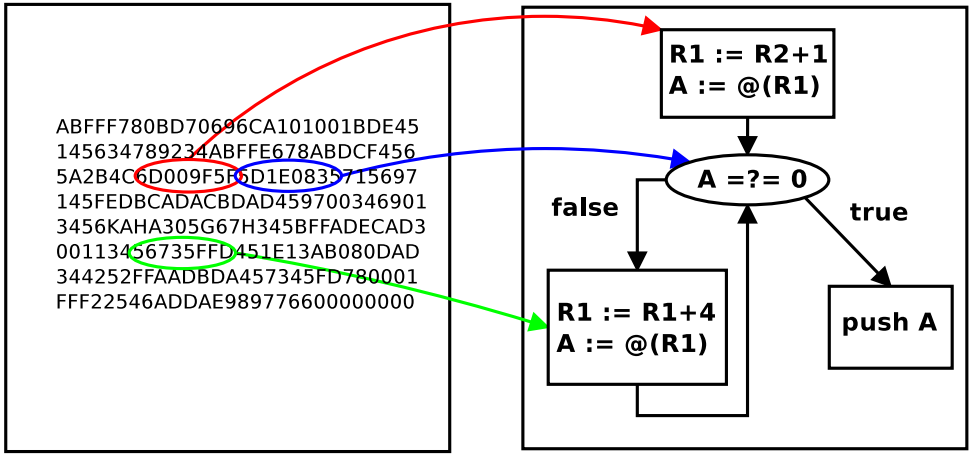
- OSMOSE achieves better binary-coverage than test suites covering source-level MCDC
- on the examples, test suites generated by OSMOSE often achieve source-level MCDC

Outline

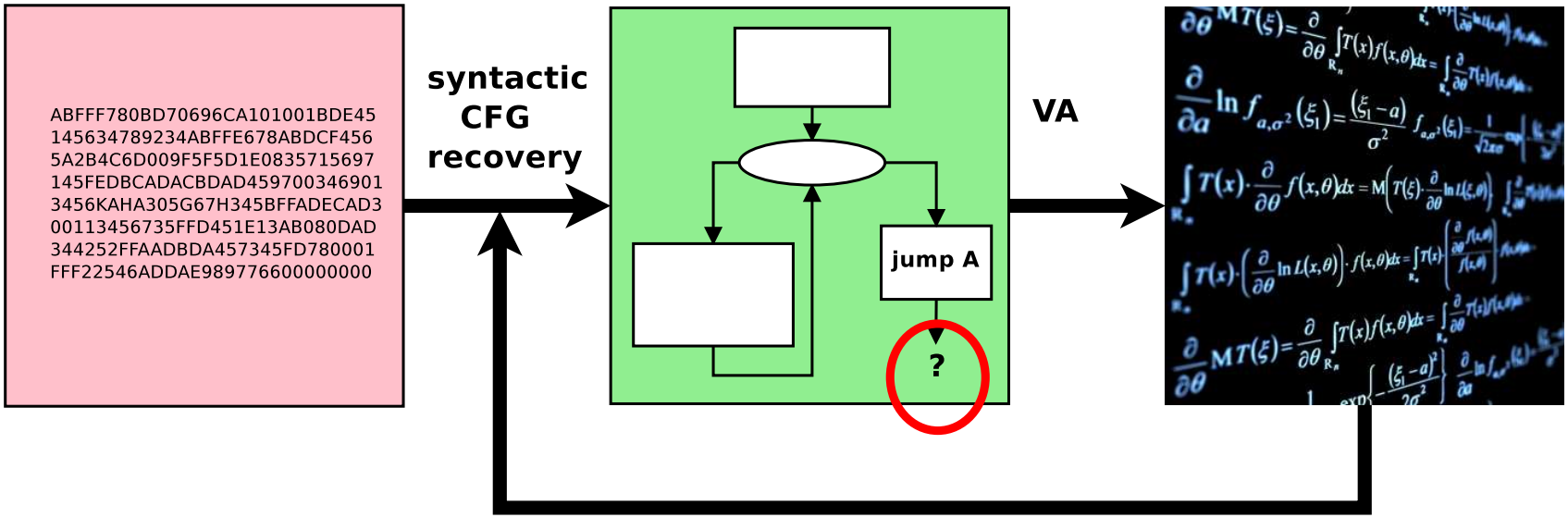
Reminder : safe CFG recovery problem

Input

- an executable code (array of bytes)
- an initial address
- a basic decoder : file $\times$ address $\mapsto$ instruction $\times$ size



Reminder : safe CFG recovery problem



"Chicken and egg" problem

Know your enemy

Dynamic jumps are pervasive [introduced by compilers]

- `switch`, function pointers, virtual methods, etc.

Sets of jump targets lack regularity

- arbitrary values chosen by compiler
- standard domains do not fit

False jump targets cannot be easily detected

- many addresses in an exec. file correspond to legal instructions

Standard domains do not fit

`jump R`, with $\mathtt{R} \in \{500, 530, 1000, 1500\}$

Stride intervals [Balakrishnan-Reps 04,05,07]

- $x \in [a..b] \wedge x \equiv c[d]$
- imprecise here : $\mathtt{R} \in [500..1500] \wedge x \equiv 500[10]$

Sets of bounded cardinality (k-sets) [Kinder-Veith 08,09,10]

- $x \in \{c_1, \ldots, c_q\}$ with $q \leq k$, or $\top$
- very imprecise if k is not sufficient : $\mathtt{R} \in \top$
- precise if k is large enough : $\mathtt{R} \in \{500, 530, 1000, 1500\}$
- precise but slow if k is too large

Our work

Key observations

- k-sets are the only domain well-suited to precise CFG reconstruction
- for most programs, only a few facts need to be tracked precisely to resolve dynamic jumps
- good candidate for abstraction-refinement

Contribution [VMCAI 2011]

- A refinement-based approach dedicated to CFG reconstruction
- The technique is safe, moreover precise and efficient on our examples

Sketch of the procedure

Abstract domain : k-sets with local cardinality bounds

- gain efficiency through loss of precision
- still a global bound *Kmax* over local bounds
- domain refinement = increase some k-set cardinality bounds

Ingredient 1 : (slightly) modified forward propagation

- propagation takes local bounds into account
- add tags to $\top$-values to record origin : $\top, \top_{init}, \top_{\langle c_1,\ldots,c_n\rangle}$
  - dedicated propagation rules : $\top_{init}$ and $\top_{\langle\ldots\rangle}$ stay in place
  - pinpoint "initial sources of precision loss" (ispl)
  - give clues for refinement

Ingredient 2 : refinement mechanism

- decide which local bound must be updated, to which value
- helped by $\top$-tags

Refinement

For each target evaluating to $\top$

- follows backward data dependencies
- follows only $\top$-values (other locations are safe until now)
- stop on initial sources of precision loss : $\top_{init}, \top_{\langle c_1,\ldots,c_n\rangle}$

How to correct

- $\top_{init}$ cannot be avoided (KO !)
- $\top_{\langle c_1,\ldots,c_n\rangle}$ may be avoided if $n \leq Kmax$ (set local bound to n)

The procedure

Procedure PaR : $(P, Kmax) \mapsto ?Invariant(P)$

pre : an unstructured program P and a global bound $Kmax$

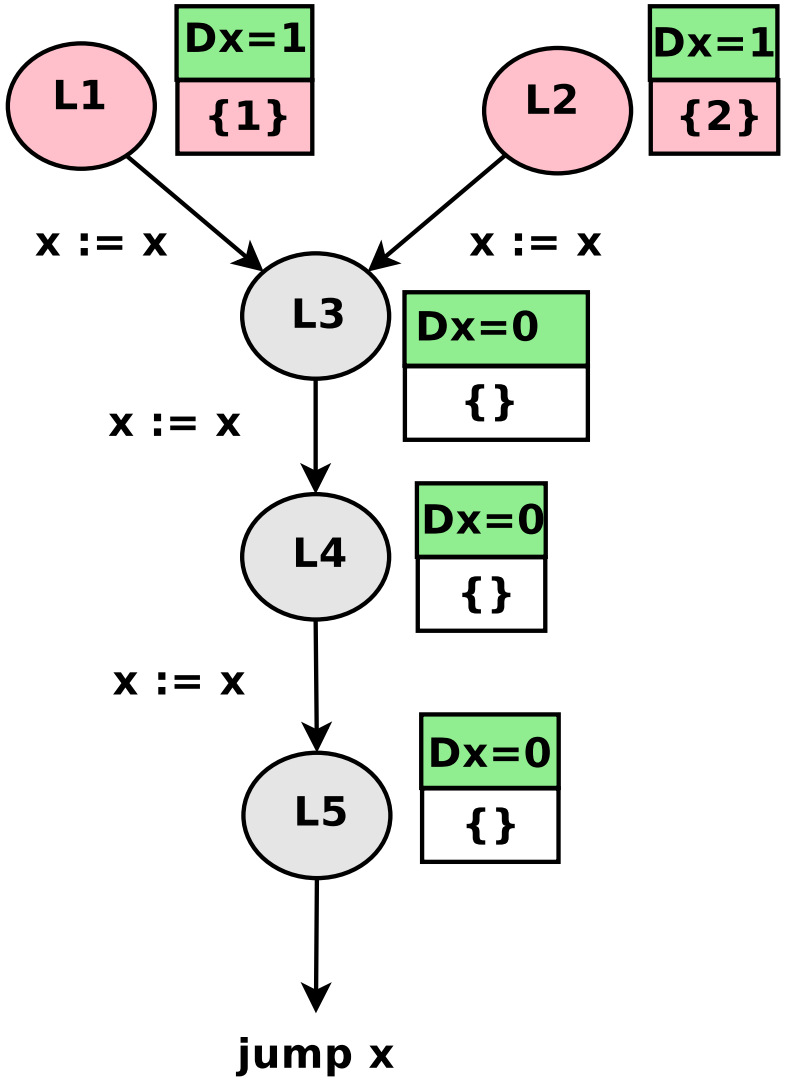
post : compute an invariant of P such that no dynamic target expression evaluates to $\top$, or fail

1. Dom $:= \{(loc, v) \mapsto 0\}$
2. forward propagate until a dynamic target exp. evaluates to $\top$
3. if no target exp. evaluates to $\top$, return the fixpoint (OK !)

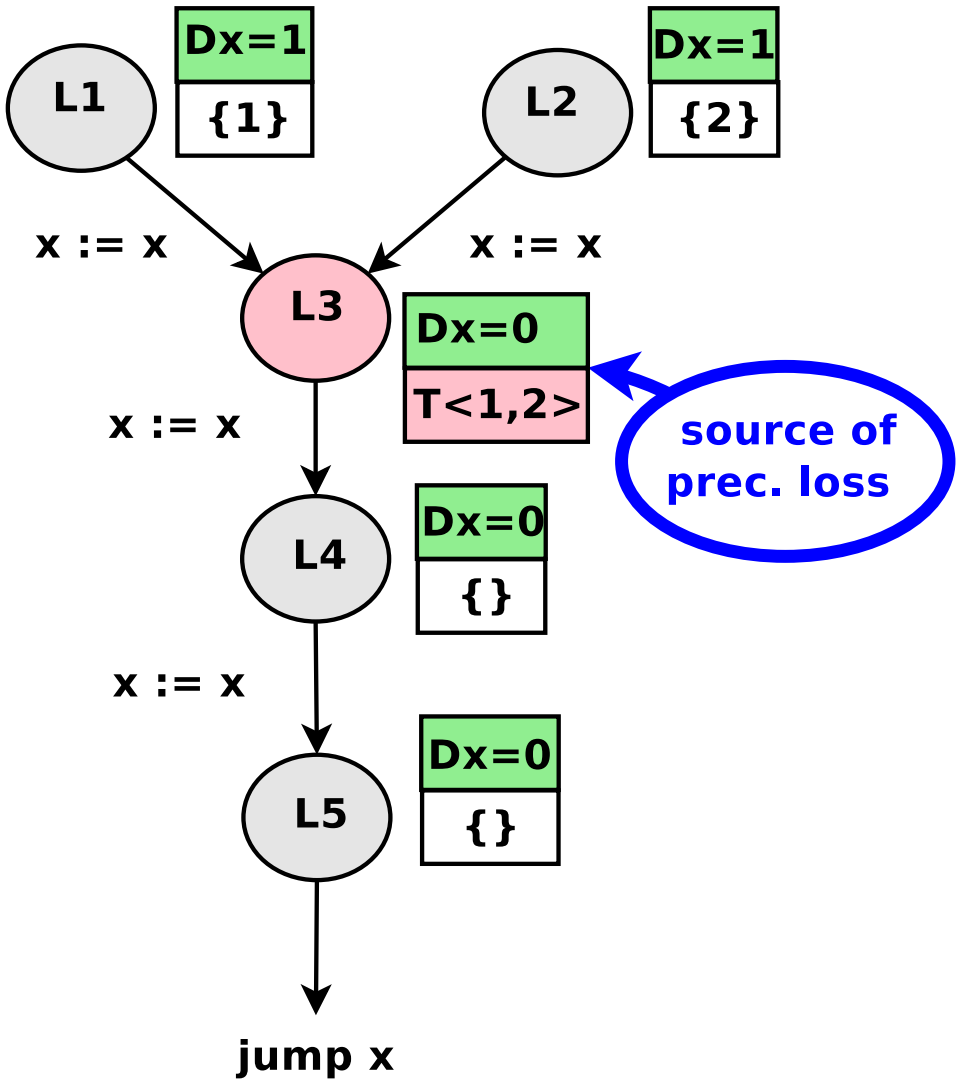
   else, try to refine the domain to avoid fault

   - if no refinement then fail (KO !)
   - else restart with refined domain (goto 2)

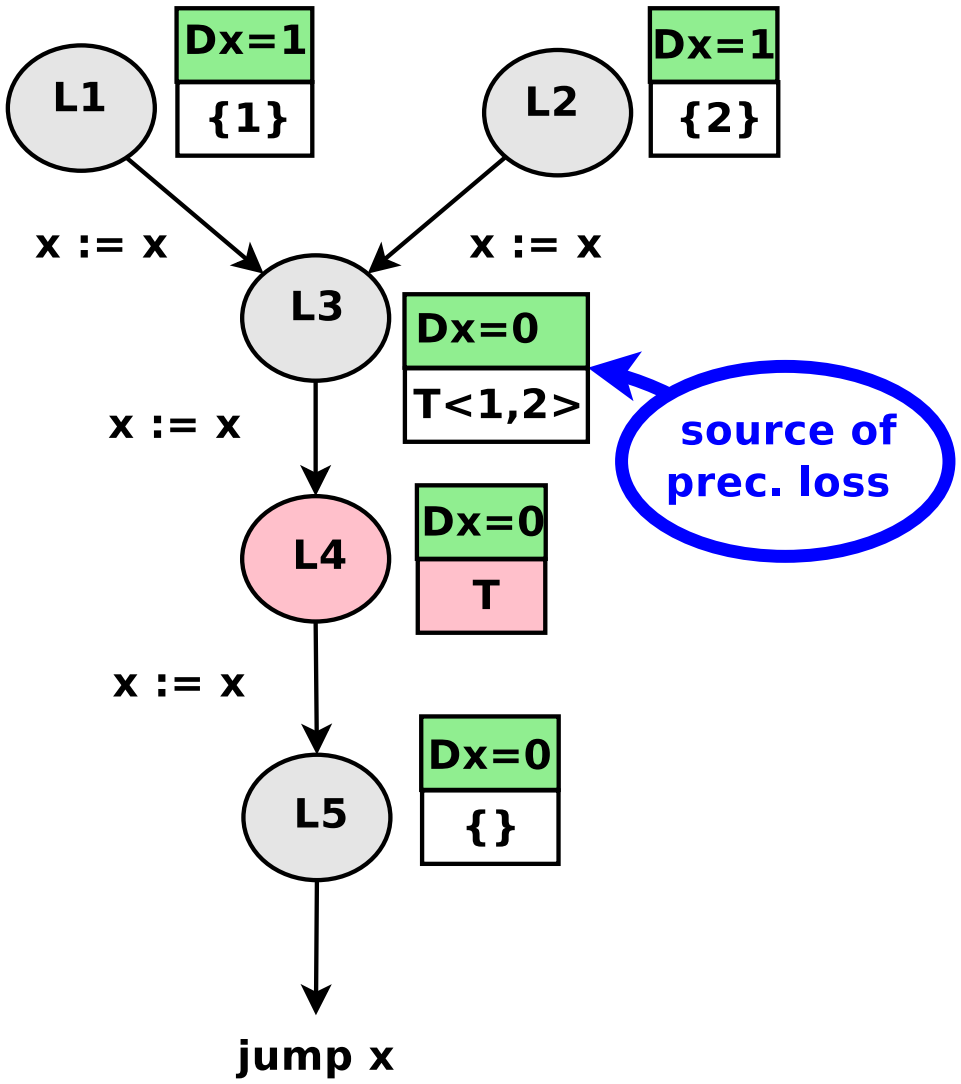
Example



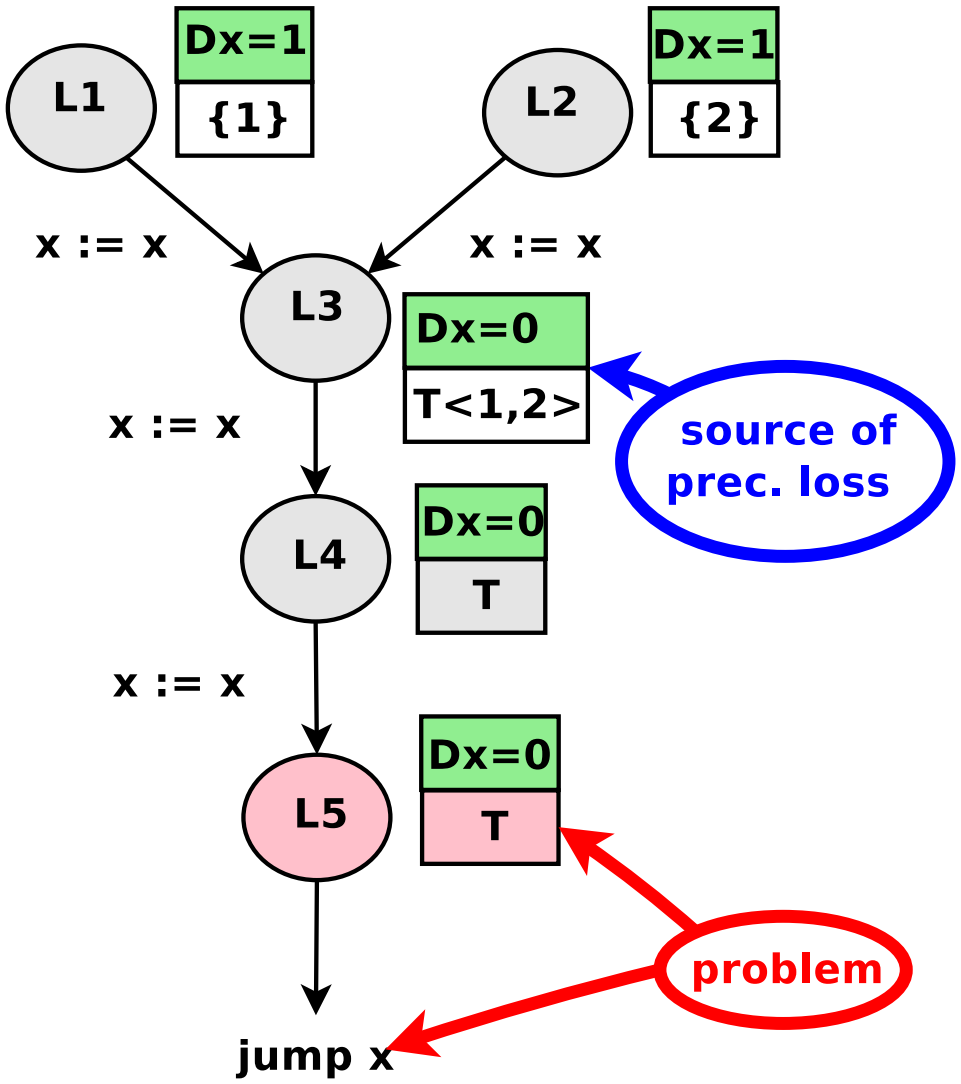
Example


L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=0
T<1,2>
source of
prec. loss
x := x
L4
Dx=0
{}
x := x
L5
Dx=0
{}
jump x

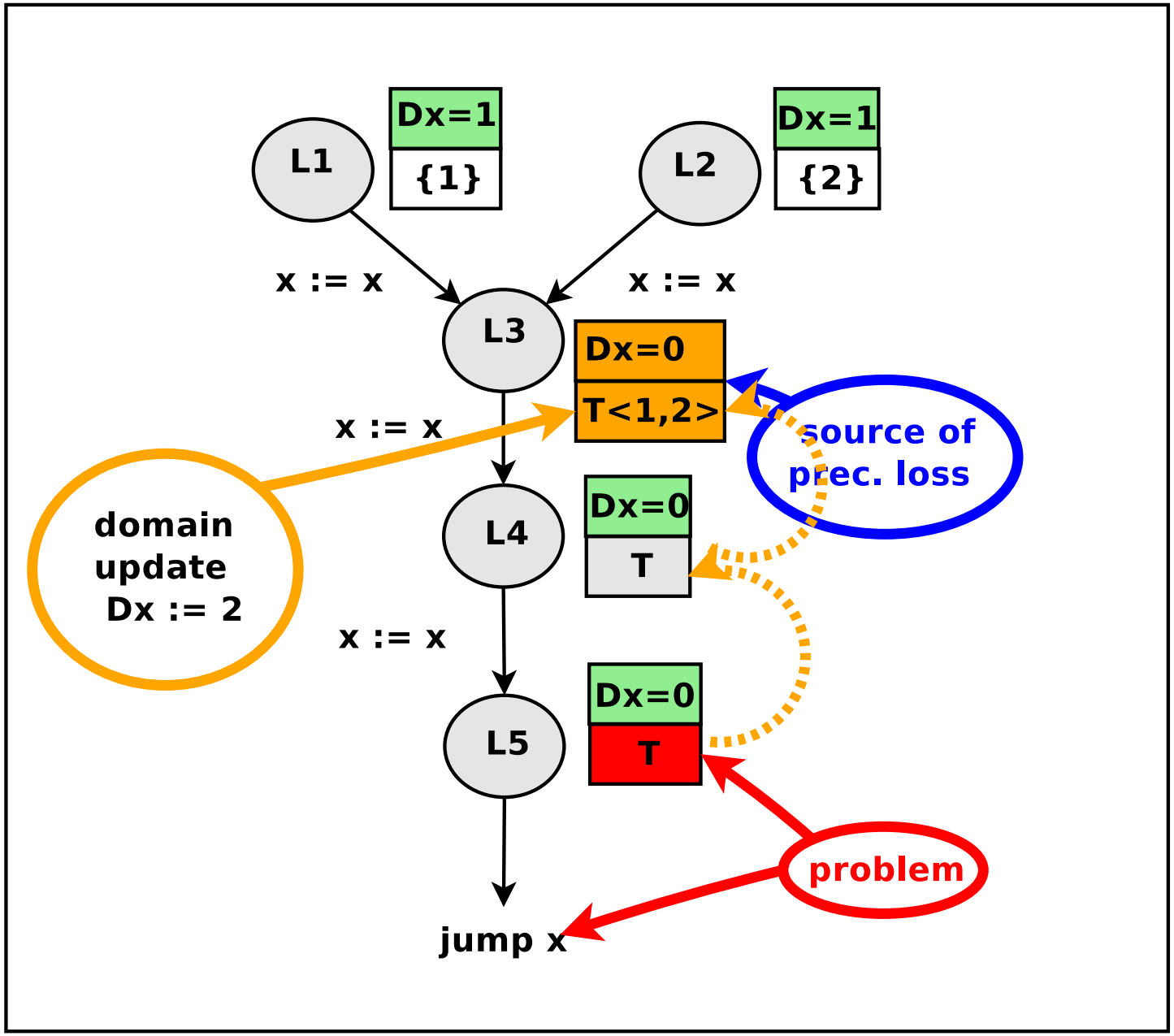
Example


L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=0
T<1,2>
source of
prec. loss
x := x
L4
Dx=0
T
x := x
L5
Dx=0
{}
jump x

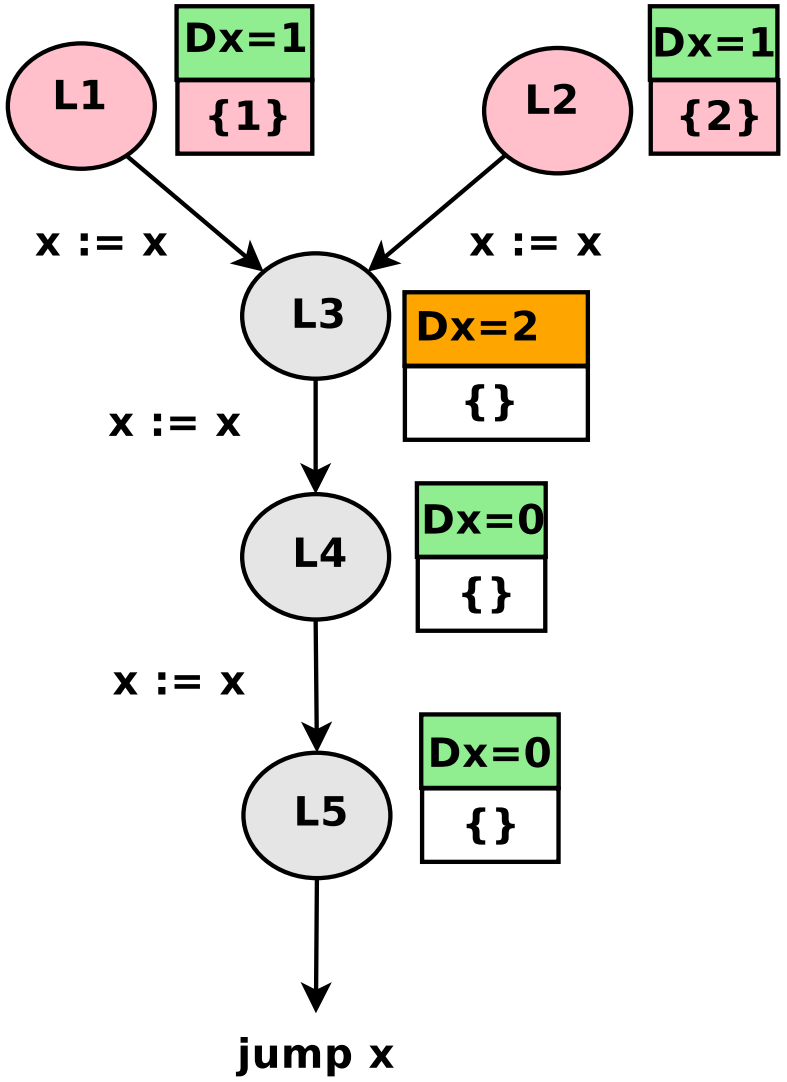
Example


L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=0
T<1,2>
source of
prec. loss
x := x
L4
Dx=0
T
x := x
L5
Dx=0
T
problem
jump x

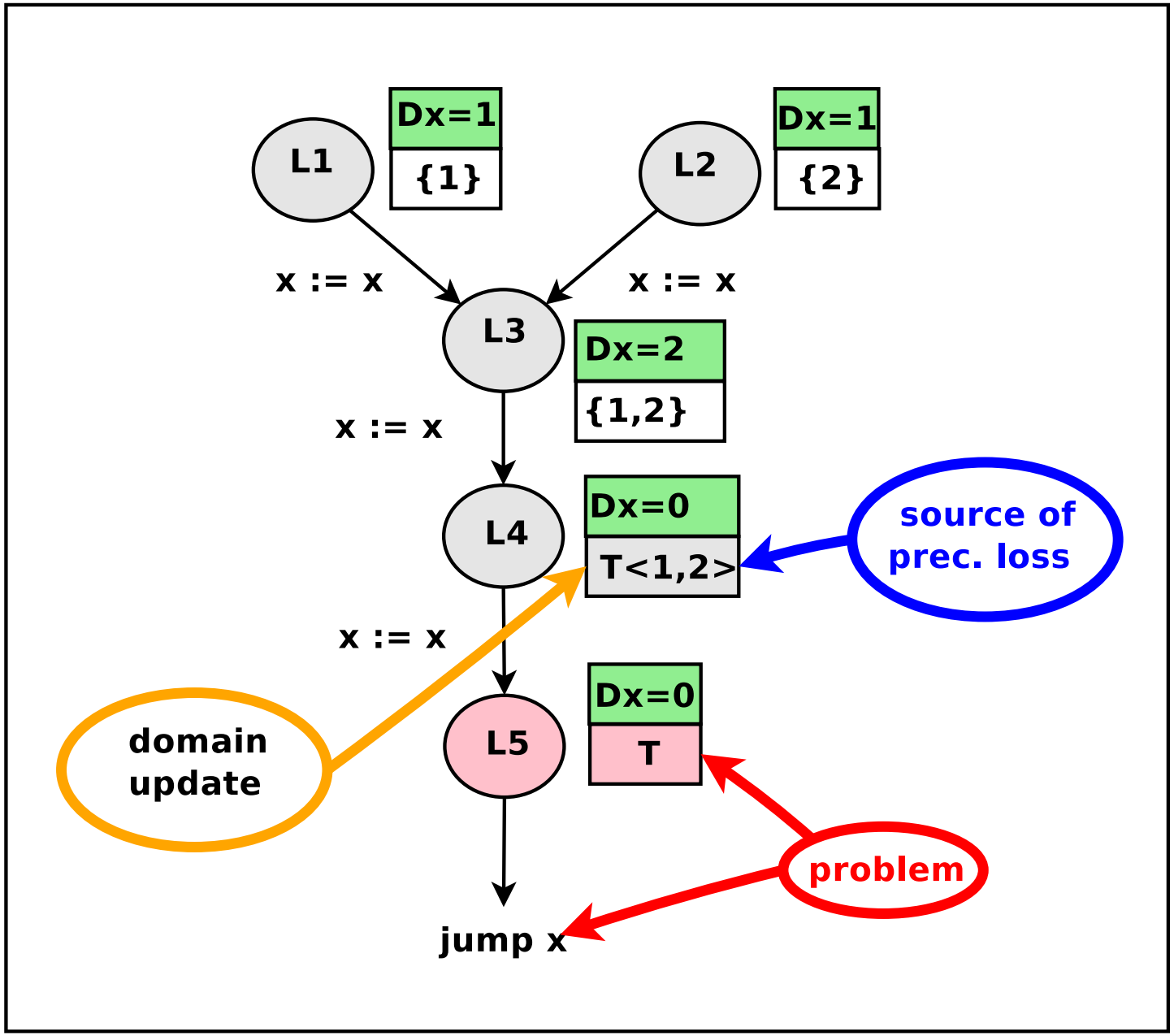
Example


L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=0
T<1,2>
source of
prec. loss
x := x
domain
update
Dx := 2
L4
Dx=0
T
x := x
L5
Dx=0
T
problem
jump x

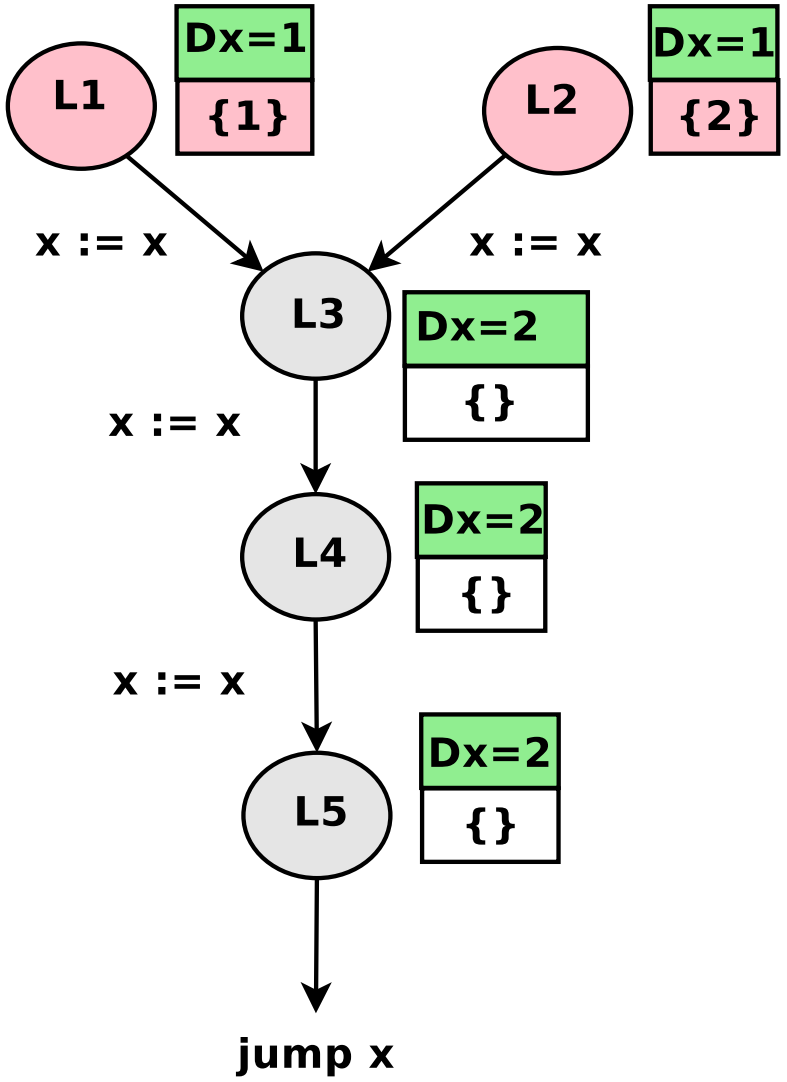
Example



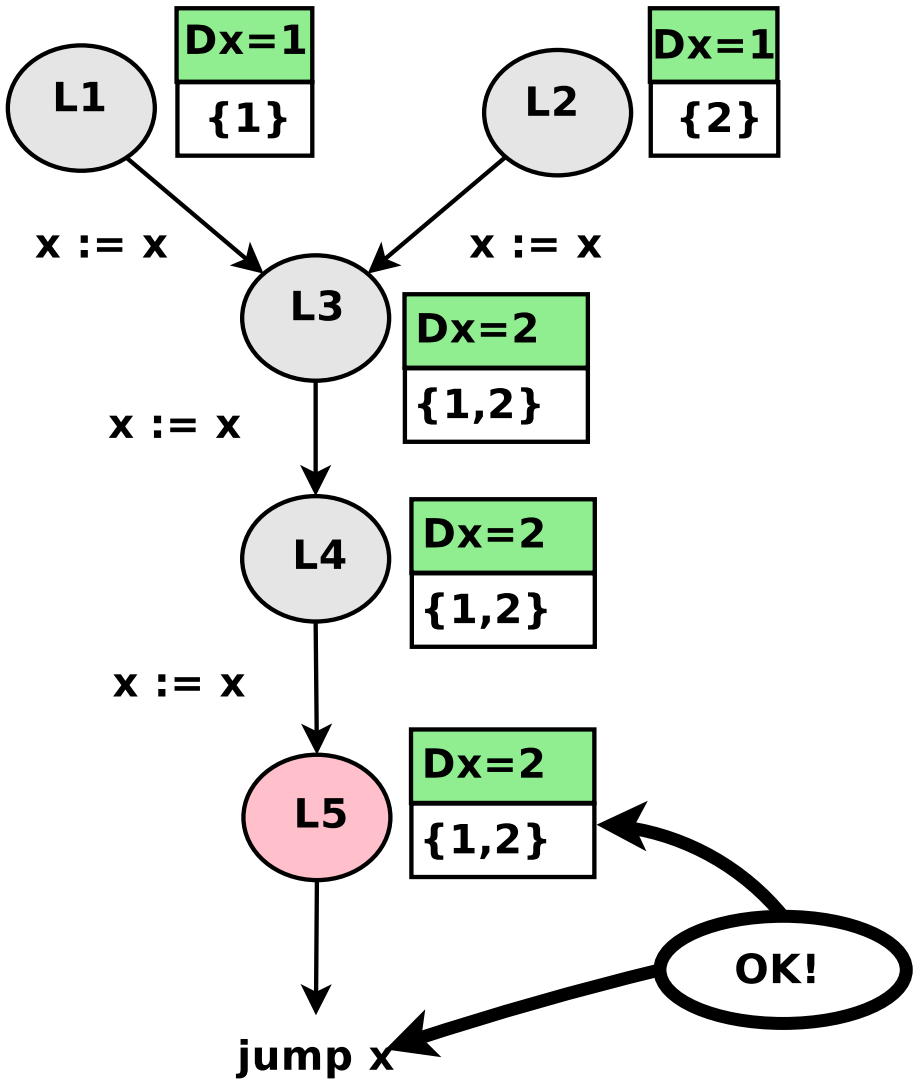
Example


L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=2
{1,2}
x := x
L4
Dx=0
T<1,2>
source of
prec. loss
x := x
L5
Dx=0
T
domain
update
problem
jump x

Example



Example



Technical details

Failure policy

- optimistic : fails only when no ispl is corrected
  [succeeds more, but more refinements]
- pessimistic : fails as soon as one ispl cannot be corrected
  [fails earlier, but may unduly fail]

Journal of the forward propagation phase

- record observed feasible branches, alias, dynamic targets
- prune backward data dependencies when searching ispl

Procedure inlining

- $\langle$ formal stack , `addr` $\rangle$
- add precision, but forbid recursion

Guarantees

Relative completeness : PaR is relatively complete if PaR$(P, \mathcal{C})$ with parameter $Kmax$ returns successfully when the forward k-set propagation with parameter $Kmax$ does.

No relative completeness in the general case mainly because of control dependencies

Relative completeness for a non trivial subclass

- non-deterministic branching
- "simple" operators ($+, -, abs, \times k$ are ok, but not $\times$)
- array indexes are required to be $\neq \top$

Fresh news !

Improved algorithm [efficiency, robustness]

- # refinements indep. of $Kmax$
- chaining of domain updates

Combination of abstract domains [precision]

- equalities : $e = e$, where $e \ ::= \ R|k|@e$
- flags : $b \Leftrightarrow e\{<, \leq, =, \geq, >\}e$
- intervals : $x \in [a..b]$

Fresh news (2)

Case 1 : compile `assume(X == Y)` into :

```
R1:=X ; R2 := Y; B := (R1==R2), assume(B)
```

- only k-sets : $B \in \{1\}$
- k-sets + equalities : $B \in \{1\} \wedge R_1 = X \wedge R_2 = Y$
- k-sets + equalities + flags : $B \in \{1\} \wedge R_1 = R_2 = X = Y$

Case 2 : prove that `@X := Y` does not affect `jump @100`

- if $X \in [101, +\infty[$, intervals ok, k-sets not ok
- requiring k-sets on write addresses might be overkill

Experiments

program	#I	#DJ	#T	max #T	#SDJ	FT	Time (sec)
aircraft	32405	51	461	16	51/51	10%	20s
SwitchCase	204	1	19	19	1/1	0%	<1s
SingleRowInput	158	1	6	6	1/1	0%	<1s
Keypad	224	1	8	8	1/1	0%	<1s
EmergencyStop	475	1	10	10	1/1	0%	17s
TaskScheduler'	171	1	5	5	1/1	0%	<1s
TaskScheduler	127	1	3	3	0/1	KO	<1s
8 puzzles	66 - 1000	21	123	9	21/21	7%	7s

I : instructions - DJ : dynamic jumps - T : feasible targets

SDJ : # dynamic jumps whose target $\neq \top$

FT : % of recovered false targets

Experiments

program	#I	#DJ	#T	max #T	#SDJ	FT	Time (sec)
aircraft	32405	51	461	16	51/51	10%	20s
SwitchCase	204	1	19	19	1/1	0%	<1s
SingleRowInput	158	1	6	6	1/1	0%	<1s
Keypad	224	1	8	8	1/1	0%	<1s
EmergencyStop	475	1	10	10	1/1	0%	17s
TaskScheduler'	171	1	5	5	1/1	0%	<1s
TaskScheduler	127	1	3	3	0/1	KO	<1s
8 puzzles	66 - 1000	21	123	9	21/21	7%	7s

I : instructions - DJ : dynamic jumps - T : feasible targets

SDJ : # dynamic jumps whose target $\neq \top$

FT : % of recovered false targets

Here : precise, efficient, tight refinement

Experiments

program	#I	#DJ	#T	max #T	#SDJ	FT	Time (sec)
aircraft	32405	51	461	16	51/51	10%	20s
SwitchCase	204	1	19	19	1/1	0%	<1s
SingleRowInput	158	1	6	6	1/1	0%	<1s
Keypad	224	1	8	8	1/1	0%	<1s
EmergencyStop	475	1	10	10	1/1	0%	17s
TaskScheduler'	171	1	5	5	1/1	0%	<1s
TaskScheduler	127	1	3	3	0/1	KO	<1s
8 puzzles	66 - 1000	21	123	9	21/21	7%	7s

I : instructions - DJ : dynamic jumps - T : feasible targets

SDJ : # dynamic jumps whose target $\neq \top$

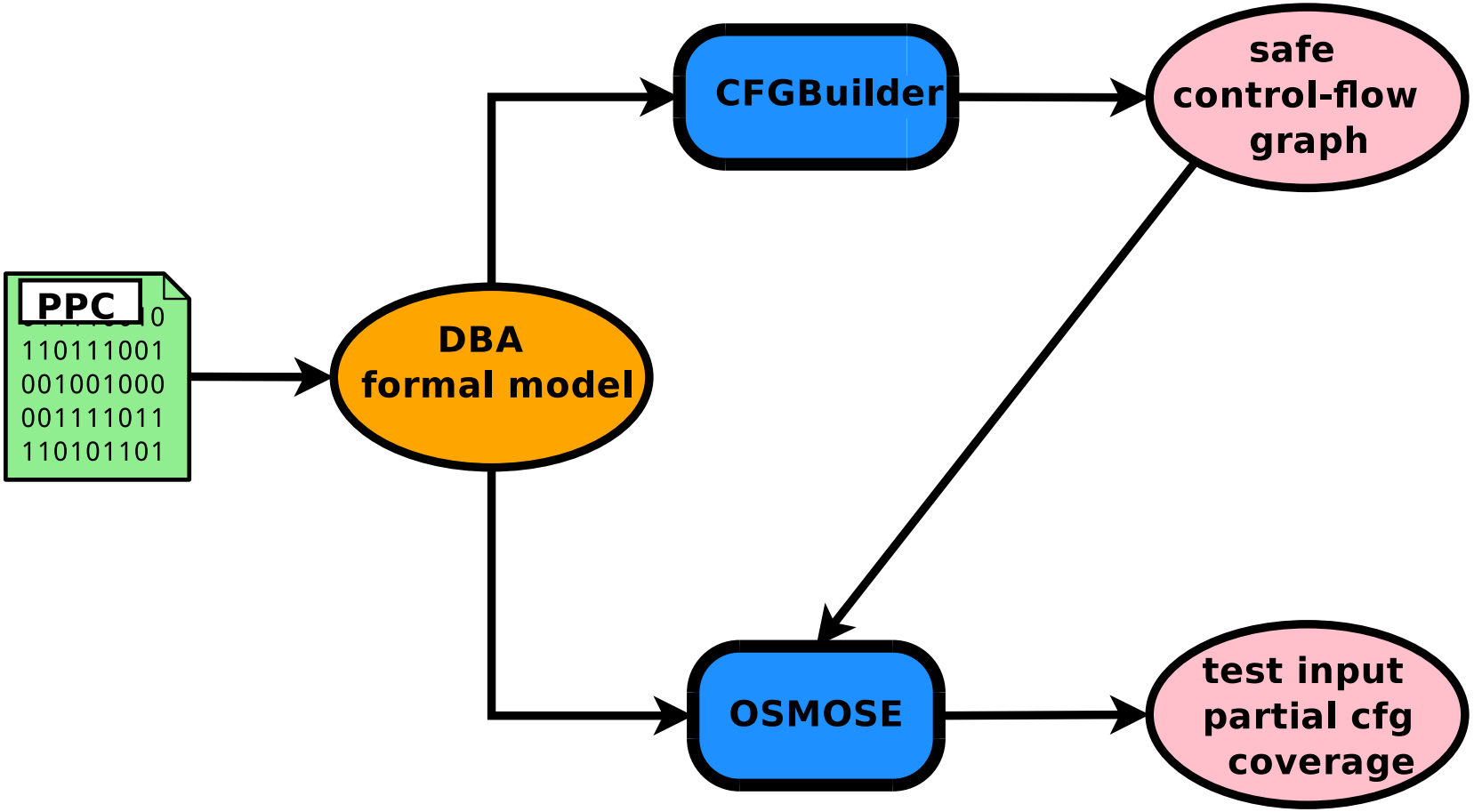
FT : % of recovered false targets

Here : precise, efficient, tight refinement

[Beware : aeronautic software easier to verify than other software]

Outline

What we have seen so far



Place in the design space

Main design choices

- stripped executable : ✓
- return address modification : ✓
- instructions overlapping : ✓
- self-modifying code : ✗
- recursion : ✗
- asynchronous interrupts : ✗

Other points

- float : ✓ CFGBuilder, ✗ CFGBuilder
- dynamic memory allocation : ●
- OS modelling : ●

On the road to security

ANR BINSEC (2013-2017) : binary-level security analysis [with LORIA]

- scale up to non-critical systems (alloc/free, libc, etc.)
- explore applications to malware & vulnerabilities

Extension of CFG recovery to non-critical executables

- dynamic mem. allocation, size++, libraries

Crash analysis

- start from a buggy but non-exploitable trace
- try to caracterize the level of exploitability of the bug

DBA extensions

- open-source platform for binary-level analysis